USENIX

MICROKERNELS and OTHER KERNEL ARCHITECTURES PROCEEDINGS

Autumn
1993

SYMPOSIUM PROCEEDINGS

Microkernels and Other Kernel Architectures

September 20-21, 1993
San Diego, California

For additional copies of these proceedings contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 U.S.A.

The price is $15 for members and $20 for non-members.

Outside the U.S.A and Canada, please add
$9 per copy for postage (via air printed matter).

Past USENIX Microkernels Proceedings (price: member/nonmember)

Microkernels Workshop April 1992 Seattle, WA $30/39

Outside the U.S.A. and Canada, please add
$20 per copy for postage (via air printed matter).

ISBN 1-880446-52-9

This volume is published as a collective work.

USENIX acknowledges all trademarks appearing herein.

Printed in the United States of America on 50% recycled paper, 10-15% post-consumer waste. ⊛

Proceedings of the USENIX Symposium on Microkernels and Other Kernel Architectures

USENIX Association

September 20-21, 1993
San Diego, California, U.S.A.

Microkernels and Other Kernel Architectures Symposium

September 20-21, 1993
San Diego California

Table of Contents

Real Time Mach

Techniques For Microkernels

Spring

Program Committee
Lori S. Grob, *Program Chair, Chorus Systèmes*
Brian Bershad, *Carnegie Mellon University*
Michael L. Powell, *Sun Microsystems Laboratories*

Is the Microkernel Technology well suited for the Support of Object-Oriented Operating Systems: the Guide Experience

R. Balter, P. Y. Chevalier, A. Freyssinet, D. Hagimont, S. Lacourte and X. Rousset de Pina

Unité Mixte Bull-IMAG/Systèmes, 2 avenue de Vignate, 38610 Gières, France
Internet: hagimont@imag.fr - Phone: +33 76 63 48 48

Abstract:
This paper describes our experience in the implementation of the Guide distributed object-oriented system on top of the Mach 3.0 microkernel. While many experimental distributed object-oriented environments have been implemented on Unix and much less on a bare machine, the emerging microkernel technology seems to provide a well suited trade-off between these two approaches. Microkernels provide modularity and flexibility for the design of a distributed operating system based on the client-server architecture, support of lightweight processes, efficient inter-process communication and the ability to implement flexible memory management policies. The goal of this paper is to provide an evaluation of the suitability of these features for the construction of distributed object-oriented operating systems.

1. Introduction

Several approaches have been considered in projects aiming at implementing a distributed object-oriented operating system. Some of them chose to build the entire system from scratch – i.e. on a bare machine (e.g. Clouds [Dasgupta 90]), but most of them have chosen to implement a layer on top of the Unix system (e.g. Emerald [Black 86], Argus [Liskov 87]). For some years a number of research groups have been experimenting the emerging microkernel technology – mainly Mach [Acetta 86] and Chorus[Rozier 88] - for building such distributed systems.

The goal of the Guide project[1] (Grenoble Universities Integrated Distributed Environment) is to provide a distributed platform for the support of object-oriented applications. The primary target applications are cooperative applications, such as multimedia document handling and software engineering, running on a set of heterogeneous workstations interconnected via a local area network.

A first phase of the project (1987-89) delivered a prototype on top of Unix. A second phase, started in 1990, intended to build a new version of the Guide platform on top of the Mach 3.0 microkernel. The purpose of this paper is to describe the main lessons learned from this experiment about the adequacy of the microkernel technology for building a distributed object-oriented operating system.

A preliminary experience with Mach 2.5 and Chorus, described in [Boyer 91], convinced us that the microkernel technology should provide a good framework for building such a system. This led us to redesign and implement a new version of the Guide system using Mach 3.0 and the OSF/1 MK server.

Guide provides a distributed virtual machine which is accessible to application programmers through object-oriented languages. Distributed computations, object sharing and persistence are key concepts of the Guide virtual machine. The Guide system provides the basic mechanisms which implement the functions of the Guide virtual machine.

Our main concern when designing this system was to define a modular and open architecture and to provide a generic support for object-oriented languages (i.e. to support different object-oriented languages such as the Guide language and a persistent distributed extension of C++). The choice of a microkernel as hosting environment was a good approach as it provides the required functions for building

[1] Guide is a component of the Comandos ESPRIT project [Balter 91].

such a system, especially as far as memory management and communication are concerned.

In the next section, we present a brief overview of the Guide model, including its object and execution model and its protection model. Section 3 briefly presents the Mach features that were used for the implementation of the system. Section 4 describes the management of execution structures. Section 5 is devoted to object management. Section 6 presents the lessons of this experiment and discusses the suitability of microkernels for system implementation. Finally, the conclusion and the perspectives are given in section 7.

2. Overview of the Guide model

In this section, we give a brief overview of the Guide model. This model is embedded in object-oriented languages such as an extended version of C++ or the Guide language. The characteristics of the Guide language may be found in [Krakowiak 90].

2.1. Object and execution model

In Guide, objects are passive, i.e. they are completely dissociated from execution structures. The execution unit is a *job* (which roughly corresponds to an "application"). A job is a potentially distributed virtual space, in which one or several *activities* (sequential threads of control) are executed. Objects are dynamically bound into a job's virtual space as a result of method calls; jobs and activities spread out to remote locations if they need to access remotely located objects. The virtual space of a job is composed of several virtual address spaces, possibly distributed on several machines; this distribution is transparent to the application. The location of the objects is determined by the system according to a location policy, currently fixed by default.

Jobs and activities communicate by means of shared objects and there is no explicit message passing. Objects are persistent (i.e. an object's lifetime is not related to that of the jobs or activities which use it). In order to start an application, a user needs to specify an initial object and an initial method. A job is then created, the initial object is bound within this job, and an activity is started by a call to the initial method of that object. Other objects are then linked into the job as needed according to the calling pattern. Applications may explicitly create new objects, new activities, and new jobs.

The object memory is organized as a two-level object store. Both levels are transparently distributed. The Virtual Object Memory (VOM) provides support for executing methods on shared, synchronized objects. The Secondary Storage (SS) provides permanent storage space for objects. The VOM acts as a cache for the SS. All objects are persistent; garbage collection is performed in the SS. Objects are named by unique system-wide identifiers, allocated at creation time.

2.2. The protection model

The protection model was defined to fulfil the following requirements:

- Ensure isolation between users (i.e. an error in a user's object must not affect the objects belonging to other users) and between applications (i.e. an error in an application must not affect applications that do not share objects with it).
- Ensure consistency with the object model. In other words, access rules must be defined in terms of the access methods applicable to objects rather than in terms of read, write or execute operations. The unit of protection is the object; in addition, the model must support users and groups.
- Solve the delegation problem. In other words, it must be possible to extend temporarily the rights of a user on a given object for the execution of a specific operation. This problem is precisely described in [Hagimont 92].

The design of the protection model relies on the following concepts:

- *User*. A user is named by the system using an identifier (*Uid*).
- *Owner*. Each object is owned by a user; ownership on a given object is inherited from that of the creator object.

In this paper, we do not describe the implementation of all these capabilities, but we focus on user and application isolation for which the use of Mach was very helpful.

3. Presentation of some Mach features

This section presents the basic Mach abstractions that were used in the implementation of the Guide system.

Kernel Abstractions

The Guide object and execution model is based on the use of the following Mach abstractions:

- *Task*: a task is a set of resources such as memory or communication ports. It can be viewed as a virtual address space divided into regions within which threads of control are executed. It provides a protected access to system resources (such as processors, ports).
- *Thread*: A thread represents a sequential activity. A thread runs within a task; there can be multiple threads running within a task, with flexible scheduling facility. Resources of a task are shared by all the threads of the task. The traditional concept of a process is represented by a single thread running within a task.
- *Port*: A port is a communication channel, which is implemented as a message queue managed and protected by the kernel. A port is only accessible via send / receive capabilities (rights).

Memory Management

The Mach kernel provides mechanisms to support large, potentially sparse virtual address spaces [Mach 92b]. Each task has an associated address map (maintained by the kernel) which controls the translation of a virtual address in the task's address space into a physical address. The physical memory is used as a cache for the virtual address spaces of tasks. The Mach kernel does not implement by itself all of this caching; some special user tasks, called *memory managers* or *external pagers*, participate in this management.

A memory manager allows a task to create *memory objects* (i.e., chunks of memory identified by ports). To address a memory object, a thread maps it within its virtual memory (i.e., the address space of its task). Once an object is mapped, page-faults on this object are treated in the same way as "normal" page faults. The kernel sends page-fault message to the memory manager to which the object belongs.

4. Execution structures management on Mach 3.0

We describe in this section the implementation of the execution structures of the Guide system on top of Mach 3.0. The first sub-section presents this mapping on Mach, using tasks and threads, and the second sub-section discusses how Mach features allowed us to fulfill the protection requirements that relates to execution structures.

4.1. Jobs and activities

The computational model of Guide defines two major entities: *jobs* and *activities*. These entities being potentially distributed, they are represented on the nodes they span by some local components.

A job is represented by a task on each node it spans (if we do not consider protection at this stage of the presentation). Activities in jobs are implemented by threads running in the tasks that implement the jobs. An activity that has visited several nodes will be represented by a thread in each task. An activity can change its execution node through a remote object call, following a synchronous scheme.

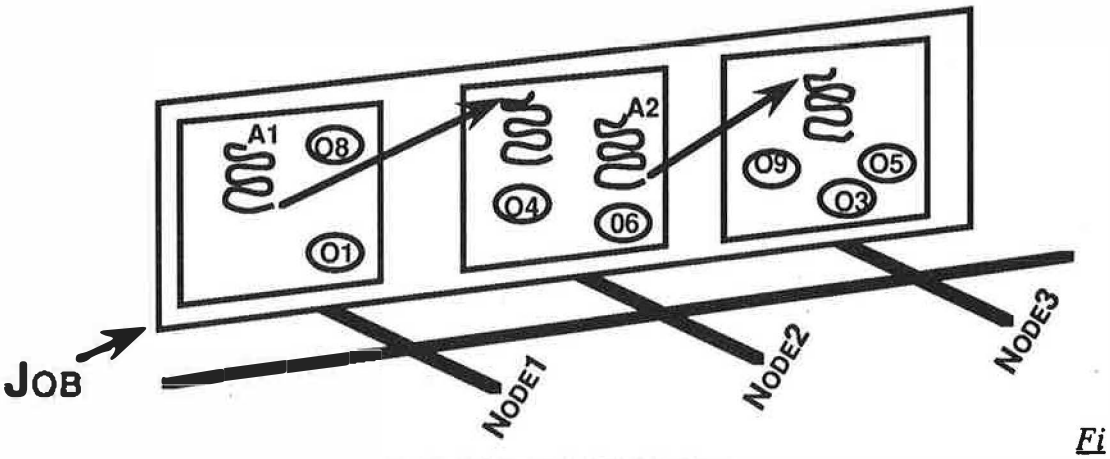


Fig. 1: Jobs and distribution

This figure shows a job running on three nodes. Activities may have spread on the three nodes. For example, activity *A1* is represented on node 1 and on node 2, while activity *A2* is represented on node 2 and on node 3. This spreading of *A1* and *A2* ensures that this job has some local components on these nodes.

Therefore, since the Guide execution model could be considered as a distributed version of the Mach execution model, Mach features were very well suited for its support.

Mach IPC is used for remote object call. A port is associated to each thread that implements a local component of an activity. This thread receives execution requests on that port from other components. At a time, there is only one active component of an activity; the others are waiting for a remote call request. For this purpose, Mach ports are very convenient because the sender of a message has no need to care about the location of the port in the network, and the interface is the same whether the task that owns the port is local or not.

Since Mach does not provide remote creation primitives, a particular daemon is present on each node to allow the creation of a local component for a job (or for an activity) on that node. This node daemon registers the communication port associated to each component of a Guide activity. The first time an activity spreads from one node to another, the daemon is queried and it returns the port to which the request must be sent. For subsequent remote invocations, the port of the target activity is cached in the task of the calling activity.

4.2. Protection

The protection requirements that relate to isolation are two-fold: to guarantee mutual isolation between jobs and to enforce user isolation despite the sharing of objects.

Jobs do not share local components (tasks) because we want to guarantee mutual isolation between jobs.

In order to enforce isolation between users, we decided that objects of different owners must be mapped in different tasks. Therefore, a job may have several representatives on a node, each of them associated to a different object owner. An activity running in an object owned by user *X* which invokes an operation on an object owned by user *Y*, crosses the task boundaries and thus is interpreted by the system. Therefore an addressing error in a method of an object can only affect objects having the same owner. A truly object-oriented protected scheme would have

required a separate task for each object, in order to prevent an error in an object from affecting another object, as it is the case in a real segmented-based operating systems like Clouds [Dasgupta 90]. This solution is not efficiently applicable here, because the average object size is small (see section 5). Figure 2 illustrates the structure of jobs including protection aspects.

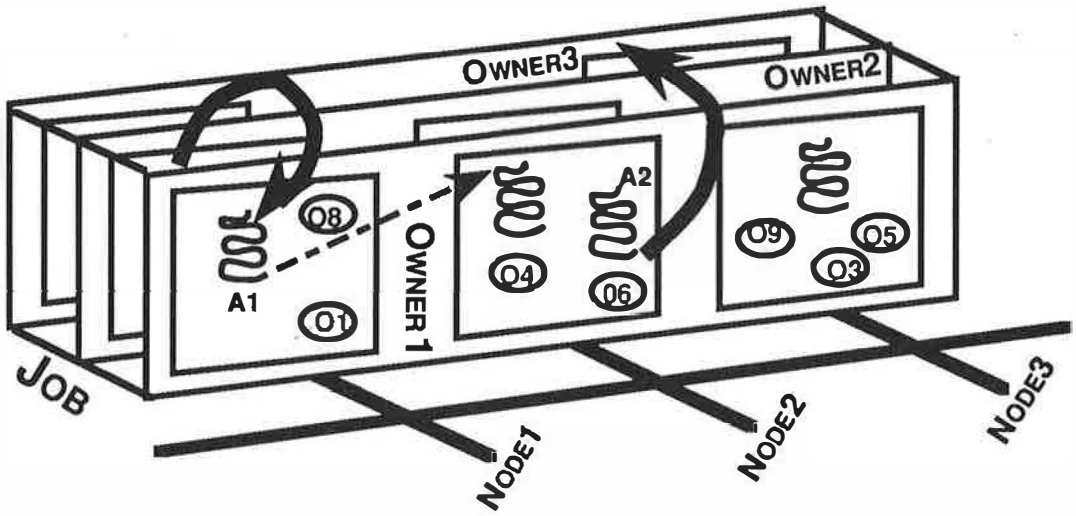


Fig. 2: Jobs and protection

This figure shows a job whose components (tasks) are distributed among three nodes. This job uses objects of different owners. These objects are respectively mapped in different tasks in order to guarantee owner's isolation. The plain arrows correspond to object calls between objects that belong to different owners.

The current implementation associates a Mach port to each local component of an activity (thread) in a task, and the thread waits for a new execution request as it is done in the distributed case.

Hence a task which is a local component of a job will only have rights on ports that are associated to local components of activities of the same job. It will not be possible for an activity to send an execution request to an activity that belongs to another job. Using ports ensures that activities belonging to different jobs will not interfere.

Another benefit of Mach ports that relates to protection appears in our solution for the delegation problem. In our solution (that could be described in a longer paper), we have to make some checks each time an object call involves two objects that belong to different owners. When such a call occurs, the object call crosses task boundaries, and the check can be done in the called task, but the problem comes from the fact that the two tasks that are involved may be associated to the same object owner on different nodes, in which case no check is needed. In fact, we need to authentify the owner associated with the calling task. In our implementation, we allocate two ports for each local component of an activity: the first one (called the *twin port*) is given to all the tasks (local component of the job) that are associated to the same object owner in the job, the second (called the *public port*) is given to all the other local components of the job. Therefore, a thread in a task that receives an object call request has to do the check if the message is received on the public port. The figure 3 illustrates this mechanism.

We also have to authenticate a task which is a local component of a job when it asks for a service to the node daemon of a node. The port that identifies a task in Mach is used as a capability in our system. A task that requests a remote invocation to the node daemon of a node authenticates itself by giving its private port (*mach_task_self*). It allows the node daemon to authenticate the job which requests a service. The same mechanism is used when a mapping request is sent to a memory

manager that will have to check if the mapping is allowed for the object owner associated to the task (see section 5).

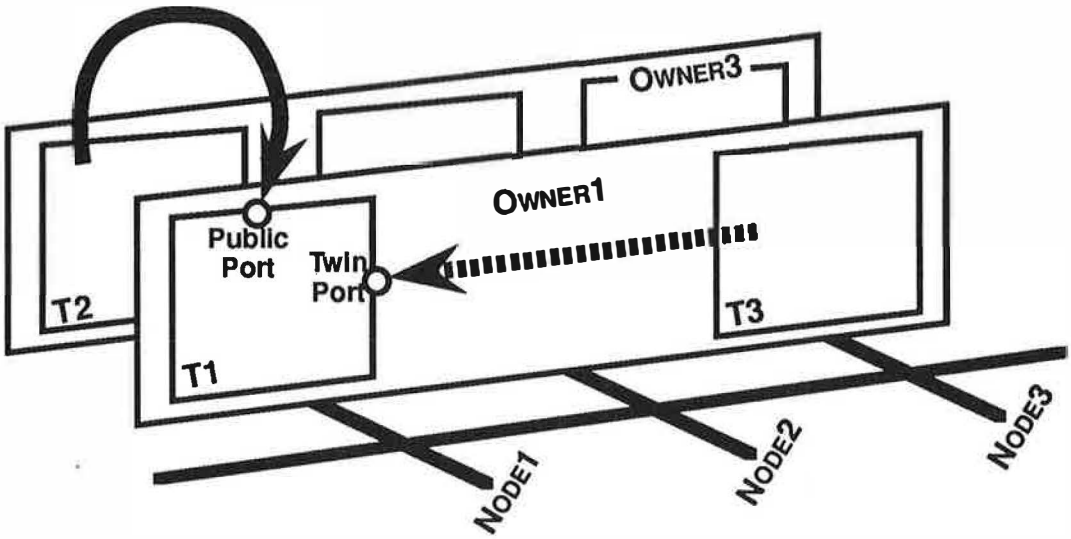


Fig. 3: the task authentication mechanism

This figure presents a job composed of five tasks. Tasks T1 and T3 contain objects belonging to Owner1, while task T2 is associated to Owner3. A method call from an object within T2 to an object within T1 will use the public port (plain arrow), while a method call from T3 will use the twin port (dashed arrow).

5. Object management on Mach 3.0

We describe in this section the implementation of the Guide Virtual Object Memory on top of Mach 3.0. The first sub-section focuses on object sharing using the memory manager facility; the second sub-section deals with persistent object storage.

5.1. Object sharing

Mach imposes two limitations for the design of the Guide system. First, Mach provides a limited number of system resources (such as ports) per task. This does not allow the potential sharing of many objects by two different jobs. Moreover, the unit of mapping in the address space of a task is a set of pages, which is much greater than the average size of our objects (a few hundred bytes). For these two reasons, we introduced the concept of *cluster* (a cluster is a set of objects) in order to support the sharing of fine-grained objects. Clusters also allow to group objects into pages in order to minimize object transfers between VOM and SS. As a cluster may contain several objects, the sharing of an object implies the sharing of other objects. Thus, all the objects in a cluster must belong to the same owner.

Microkernels provide the *memory manager* facility. A memory manager is a user process which allows other processes to map a chunk of memory, identified by a port, into their virtual address spaces. From the microkernel point of view, the memory manager is responsible for handling page faults that may arise on this chunk of memory. We used this facility to implement clusters. Clusters are managed by a set of memory managers that we call *cluster managers*. These managers are distributed among the network and may cooperate together. A cluster manager implements two kinds of functions:

- mapping, sharing and protection control;
- paging functions such as page-in/page-out requests from the Mach kernel (the interface between the kernel and the cluster manager is defined in [Mach 92a]).

Figure 4 depicts the interactions between jobs, the microkernel and the cluster managers.

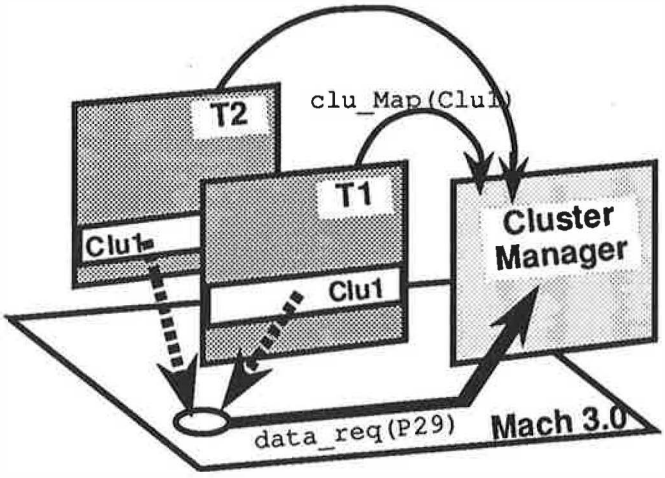


Fig. 4: Cluster mapping

This figure depicts the sharing of the cluster *Clu1* between tasks *T1* and *T2*. In order to map that cluster, both tasks send a *clu_Map* request to the cluster manager. After checking the rights of the tasks, the cluster manager returns a port (P29) which will be used by the tasks when mapping the cluster. Access to the cluster will cause page faults that will be redirected by the kernel to the cluster manager using the port P29.

5.2. Object storage

In the computational model defined above, we made the distinction between two levels of memory: the VOM which provides support for method executions and which consists of the union of physical memory and the swap disk spaces of each node, and the SS, which provides long term storage support. The interface between VOM and SS is defined in terms of load and store operations. The separation between VOM and SS is useful for the following reasons:

- Modularity (i.e., ability to reuse existing storage servers such as AFS or Pegasus [Leslie 93]),
- Security (i.e., access control to the persistent storage interface),
- Quality of service (i.e., servers can provide different storage policies such as reliability, fast access, etc.).

Moreover, movements of clusters between the VOM and the SS are entirely managed by cluster managers. This allows this transfer to be performed very efficiently. Transfers are performed on demand (i.e. the whole cluster is not transferred in VOM at mapping time): a page is only loaded in VOM when the cluster manager receives a fault on this page. This ensures that the transfer occurs only for those pages that are actually used by jobs. Furthermore, when the cluster manager has to store the modified image of a cluster in SS, only dirty pages are copied to SS. This scheme allows the traffic between VOM and SS to be reduced.

6. Evaluation

In this section, we summarize the lessons learned from our experience in building the Guide system on top of Mach 3.0. We first discuss the adequacy of microkernels for the implementation of our system. Then, we present some performance figures performed on our prototype.

6.1. Adequacy of Mach 3.0

We have shown in the description of the implementation of the Guide system that the microkernel technology provides a well suited platform for the design of a distributed object-oriented operating system.

As mentioned above, the Guide model can be viewed as a distributed version of the Mach model. Consequently the mapping of the Guide abstractions (jobs and activities) on the Mach abstractions (tasks and threads) was straightforward. In Guide, an application involves a potentially distributed virtual address space called a job within which several activities may run. Jobs and activities are naturally represented by a set of tasks and threads running on the nodes visited by the application.

The benefits of the port concept are twofold:

- The port provides a location transparent address for message passing between tasks. Therefore, in our implementation, we were able to develop and debug the entire Guide kernel on a single machine, since message passing between tasks on a single machine or on different machines have the same interface. The step between a centralized prototype and the fully distributed version was quite small compared to the former experiment achieved on top of Unix.
- Mach ports are protected in the sense that a port cannot be used unless it has been explicitly given by a task that has the required rights on this port. This allows the implementation of the isolation requirements for jobs and users. This also allows the authentication of these user-level tasks when they cooperate with the Guide kernel.

The ability to design our own memory manager was one of the key benefits from using the microkernel approach:

- It allows a simple implementation of object sharing between different nodes, which was not straightforward on Unix.
- It allows the implementation of flexible consistency policies according to application requirements.
- It allows the efficient management of fine-grained objects. The use of memory managers allows a clear separation between the object as unit of addressing, the cluster as unit of mapping, and the page as unit for I/O transfers. This allows an optimization of the multiple facets of memory management.
- Cluster managers enforce the protection model by controlling user rights on objects into a protected server. In addition, clusters are not made visible to applications. This also improves the protection scheme of the system.

However the implementation of Guide suffered from some missing features which are listed below:

- The ability to create a task on a remote node would greatly simplify the overall architecture and especially the management of the execution structures.
- Protected ports have a major drawback in a distributed environment: applications cannot share ports. Should this facility be available, finding a specific task within a job would have been improved, for example by removing the need to contact the node daemon.
- Port groups would have been convenient for managing distributed entities such as jobs and activities. They would also have been useful for providing fault tolerance facilities.
- Implementation of synchronisation tools such as semaphore objects shared between task located on the same or different nodes is not an easy work. The designer should either provide a semaphore server or ensure that all threads sharing a semaphore object know each other's ports.
- As proposed in [McNamee 90], it would be interesting to provide the ability to manage page replacement in physical memory at the level of a cluster manager, since a cluster manager does have some knowledge about cluster usage that the kernel cannot manage. For example in the current

implementation, cluster managers collect and store information about page access or cluster sharing which could allow improved paging.
- It is not possible currently to develop servers on Mach independently of OSF/1 (for example to get hight level I/O functionalities).

6.2. Performance

The implementation of Guide-2 started at the end of 1991, using first Mach-3.0 and then Mach 3.0/NORMA both with OSF-1 /MK-13 on Bull-Zenith P.C. i486 (33 MHz) connected to a 10 Mb Ethernet. Mach-3.0/NORMA is a version of Mach-3.0 that allows to consider a set of workstations connected by a LAN as a multiprocessor. NORMA integrates the network communication inside the microkernel, with a substantial gain of performance (a factor of about 50). Table 1 gives some preliminary figures. These figures measure the elapsed time of the basic functions provided by the cluster manager: cluster creation, cluster mapping, handling a read or write page fault and cluster unmapping. In this experiment clusters in secondary storage were simply implemented by Unix files. A more elaborate version of the storage server is under way.

Clusters Operations	*Time (in ms)*
clu_Create[1]	130,0
clu_Map	10,0
page_read[2]	6,8
page_write	6,8
clu_Unmap[3]	124,0

Creating a cluster is a cost intensive operation since it requires to create two Unix files: one for storing the cluster descriptor and one for the actual cluster. In addition, this operation involves an access to a specific file containing the information used to allocate a global unique name for the cluster. The cost of this operation will be significantly reduced when using the final version of the storage server.

Mapping a cluster requires to read the descriptor of the cluster in order to pick up its size and its owner. In the upcoming version of the storage server cluster descriptors will be cached in main memory to reduce the overhead of accessing the cluster descriptor.

The figures related to page faults (*page_read* and *page_write*) take into account the cache management associated to the Unix file system.

Cluster unmapping implies to store all modified pages in SS. In our scenario all pages were modified; this explains the relatively high value for this figure. Real applications are not expected to update all pages at each execution; if it were the case, the cluster manager would obviously become a bottleneck for the whole system.

To conclude this section, we are convinced that these results are very promising regarding the level of functionality provided by the cluster machine (i.e.;, object sharing, persistence and protection) and the benefits, in terms of security and modularity, offered by our architecture. The performance is expected to be slightly improved when the implementation of a full storage server will replace the current implementation based on Unix files.

7. Conclusion

We have shown in this paper how we used Mach 3.0 features in the implementation of the Guide system. The Guide system provides an execution

[1] Clusters are 10 pages long (40960 bytes).

[2] Pages are transfered from SS to VOM (in read or write case).

[3] Pages are stored from execution memory to persistent store.

environment for an object-oriented programming language. The main features of the system are: persistent shared objects, supported by a two-level distributed storage, transparent distribution of objects, execution model based on concurrent, distributed jobs and activities, and support for protection.

The objective of this paper was to assess our three year experience in using Mach and to present the lessons learned from this work concerning the adequacy of microkernels for building distributed systems. Systems like Mach and Chorus are organized as a set of servers which are managed by a minimal microkernel. The flexibility provided by this architecture greatly helps the design of distributed systems. Furthermore modularity allows different parts of the system to be designed and tested separately on the one hand, and various resource management policies to be experimented on the other hand.

The Guide prototype is currently running on a network of i486-based machines using Mach 3.0 and the OSF/1 MK server. The implementation is nearly complete. Several applications such as a distributed co-operative spreadsheet are already running, that allow debugging and tuning of the system. These experiments have shown that the standard version of Mach 3.0 performs poorly over the network. Therefore, in co-operation with OSF-RI, we are currently experimenting the NORMA version of Mach 3.0 to reduce the cost of remote invocations. Preliminary experiments with this advanced version are very encouraging.

Acknowledgments. We would like to thank Professor Jacques Mossière for his help in reviewing this paper. The Guide project is supported by the Commission of European Communities through the ESPRIT project COMANDOS (Construction and Management of Distributed Open Systems), the Universities of Grenoble (Institut National Polytechnique de Grenoble – Université Joseph Fourier) and Centre National de la Recherche Scientifique.

The Guide implementation on top of Mach 3.0 was carried out in collaboration with the OSF-Research Institute (Grenoble).

Bibliography

[Acetta 86]
M. J. Acetta, R. Baron, W. Bolowsky, D. Golub, R. Rashid, A. Tevanian, M. Young, Mach : a new kernel foundation for Unix Development, *Proc. of the USENIX 1986 Summer Conference*, Jul. 1986, pp. 93-112

[Balter 91]
R. Balter, J. Bernadat, D. Decouchant, A. Duda, A. Freyssinet, S. Krakowiak, M. Meysembourg, P. Le Dot, H. Nguyen Van, E. Paire, M. Riveill, C. Roisin, X. Rousset de Pina, R. Scioville, G. Vandôme, Architecture and implementation of Guide, an object-oriented distributed system, *Computing Systems*, vol. 4, n° 1, 1991, pp. 31-68

[Black 86]
A.P. Black, N. Hutchinson, E. Jul, H. Levy, Object structure in the Emerald system, *Proc. First ACM Conf. on Object-Oriented Systems, Languages, and Applications (OOPSLA)*, Portland, Sept. 1986

[Boyer 91]
F. Boyer, J. Cayuela, P. Y. Chevalier, A. Freyssinet and D. Hagimont, Supporting an object-oriented distributed system: experience with Unix, Mach and Chorus, *Proc. Symposium on Experience with Distributed and Multiprocessor Systems*, Atlanta, Mar. 91, pp. 283-299

[Dasgupta 90]
P. Dasgupta, R.C. Chen, S. Menon, M.P. Pearson, R. Ananthanarayanan, U. Ramachandran, M. Ahmad, R.J. LeBlanc, W.F. Appelbe, J.M. Bernabeu-Auban, P.W. Hutto, M.Y.A. Khalidi, and C.J. Wilkenloh, The Design and Implementation of the Clouds Distributed Operating System, *In Computing Systems*, vol. 3, n° 1, Winter1990, pp. 11-45

[Hagimont 92]
D. Hagimont, S. Krakowiak and X. Rousset de Pina, Protection in an Object-Oriented Distributed System, *Proc. of the 4th Int. Workshop on Object Orientation in Operating System*, Paris, Sep. 1992

[Krakowiak 90]
S. Krakowiak, M. Meysembourg, H. Nguyen Van, M. Riveill, C. Roisin, and X. Rousset de Pina, Design and implementation of an object-oriented strongly typed language for distributed applications, *Journal of Object-Oriented Programming*, vol. 3, 3, pp 11-22

[Leslie 93]
I. M. Leslie, D. McAuley and S. J. Mullender, Pegasus - Operating System Support for Distributed Multimedia Systems, *ACM Operating Systems Review*, 27(1), Jan. 1993, pp. 69-78

[Liskov 87]
B. Liskov, D. Curtis, P. Johnson, and R. Scheifler, Implementation of Argus, *In Proc. 11th Symp. on Operating System Principles*, vol. 5, 21, Nov.1987, pp. 111-122

[Mach 92a]
Mach 3 Kernel Interfaces, *Open Software Foundation and Carnegie Mellon University*, edited by K. L. Loepere, Jan.1992

[Mach 92b]
Mach 3 Kernel Principles, *Open Software Foundation and Carnegie Mellon University*, edited by K. L. Loepere, Jan.1992

[McNamee 90]
D. McNamee and K. Armstrong, Extending the Mach External Pager Interface to Accomodate User-Level Page Replacement Policies, *Proc of the Mach Usenix Workshop,* Burlington, VE, Oct 1990, pp 31-43

[Rozier 88]
M. Rozier, V. Abrossimov, F. Armand, J. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, P. Leonard, S. Langlois and V. Neuhauser, Chorus Distributed Operating Systems, *Computing Systems*, vol. 1, n° 4, 1988, pp. 305-370

Object Oriented Transaction Processing in the KeyKOS Microkernel

William S. Frantz
Periwinkle Computer Consulting
16345 Englewood Ave.
Los Gatos, CA USA 95032
frantz@netcom.com
408/356-8506

Charles R. Landau
Tandem Computers Inc.
19333 Vallco Pkwy, Loc 3-22
Cupertino, CA USA 95014
landau_charles@tandem.com
408/285-6148

Abstract

Three major technological directions in computer technology are transaction processing, object orientation, and microkernel operating systems. The KeyKOS operating system and the KeyTXF transaction processing system combine all three of these technologies. The design of KeyKOS directly provides operating system level objects on a microkernel base. In order to maintain the integrity of these objects, KeyKOS takes periodic checkpoints of the entire system. In addition, KeyKOS provides facilities for transaction processing which achieve very high transaction rates. Object oriented technology facilitates construction and reuse of transaction applications. This paper describes how these ideas are combined in the KeyKOS system.

1 Introduction

This paper examines the structure of an application environment that combines three technologies: transaction processing, object orientation, and microkernel operating systems. The KeyTXF transaction processing system, which runs on the KeyKOS microkernel operating system, provides high performance object oriented transaction processing. We believe that this work demonstrates that these technologies can be successfully combined, gaining the advantages of each.

1.1 Transaction Processing

The technology of transaction processing significantly reduces the effort required to build applications by allowing applications to combine a group of one or more operations on one or more objects while providing the following "ACID" properties:

Atomicity - "all or nothing"
Consistency - The application can ensure that the data base state does not violate application defined consistency rules.
Isolation - Transactions are logically executed serially.
Durability - Results of a completed transaction are never lost.

1.2 Object Orientation

Object oriented systems support encapsulation and isolation of function. They support the association of data with the algorithms that manipulate that data and the protection of that data from other algorithms.

They make it easy to change the implementation of functions without affecting the users of those functions. They make it easy to build new function using existing functions as building blocks.

1.3 Microkernel

Microkernel systems support easy extension of "system" facilities. They permit support of more than one application programming interface. They significantly reduce the effort required to port the system to a new hardware architecture. They increase reliability by decreasing the amount of code whose failure crashes the entire system.

2 Related Work

2.1 Persistent Memory

Satish M. Thatte describes a design for a persistent virtual memory system [9] with transactions for object-oriented garbage collected programming systems and object-oriented databases. In contrast with the approach to persistent memory used in KeyKOS, his design requires unforgeable pointers, bounds checking, and automatic garbage collection. His transaction manager keeps an external redo log outside virtual memory, while KeyTXF keeps the log in virtual memory.

2.2 QuickSilver

QuickSilver [8] is an operating system with a small kernel that supports extensive use of transactions. QuickSilver is not explicitly object-oriented. In contrast, KeyKOS uses object technology to support transactions without requiring awareness of transactions in the microkernel.

2.3 X/Open and OSI Distributed Transaction Processing

These standards address the need for defined interfaces between a transaction application and the software components with which it interacts. Though not specifically object-oriented, such an interface can be viewed as an object interface, whether the object is local or remote. This interface makes explicit use of Transaction ID's.

2.4 Object Management Group

The Object Management Group has identified the Object Transaction Service as a service which may be the subject of a future Request For Proposal. This work is in a very preliminary stage.

3 KeyKOS Features Important to Transaction Processing

KeyKOS® [2 and 4] is a fault tolerant, object-oriented, microkernel operating system. KeyKOS uses capabilities[6] to address objects. KeyKOS provides operating system level protection to its objects.

The checkpoint mechanism [5] is the principal way KeyKOS makes data durable or persistent. The system level checkpoint periodically saves the entire state of the system on non-volatile storage (i.e. disk). This checkpoint protects all processes, data, files, registers, etc. against the system going down (whether planned or unplanned). When KeyKOS is restarted, the state of the system at the most recent checkpoint is restored,

and the all processes continue running from that state. Taking a checkpoint interrupts execution of processes for less than a second. Restart requires only the time for active processes to fault in their working sets.

A system level checkpoint is taken every few minutes, so the system must be able to commit transactions between checkpoints. To support the durability requirement of transaction processing, KeyKOS has an additional kernel-implemented operation, the Journalize Page operation. This operation saves the current contents of a virtual memory page to disk immediately so that the data will not be lost. After a restart, the data in the page will represent the data at the time the Journalize Page operation was called (or at the previous checkpoint, whichever is more recent). Using the Journalize Page operation, the KeyTXF transaction processing system can fail at any point in time without losing any committed transactions.

Fault tolerance was a major goal of the KeyKOS microkernel. In addition to the checkpoint and journaling mechanisms, the microkernel uses automatic disk mirroring to ensure that no data can be lost from a single disk failure at any time. A special mechanism, described in section 6, is used to handle disk failures that occur in the middle of writing a disk page, resulting in the page being partially written.

KeyKOS uses a time stamp which allows it to record a time with a resolution of better than a microsecond and a date covering over 100 years[1]. This time stamp is used extensively in KeyTXF. KeyTXF requires that time stamps never go backwards, and that the resolution is sufficient to generate distinct time stamps at the desired transaction rate.

KeyKOS maintains the time of the last restart and the time of the last checkpoint in a special page called the checkpoint page. This information is used by KeyTXF to detect when a restart has occurred.

The Compare and Swap operation atomically compares an "old" value with a value in memory. If they are the same, it stores a "new" value into that memory location. It returns an indication of whether a store occurred. If Compare and Swap is not available in hardware, it may be implemented in the kernel. KeyTXF assumes that it is atomic in relation to checkpoint/restart.

KeyTXF is implemented directly on the microkernel interface. KeyTXF applications can be implemented in any of the supported operating system environments including the microkernel.

4 KeyTXF Structure

A transaction processing application is viewed as an object which communicates through references to other objects. On the "front end", the application communicates with requesters of transactions. On the "back end", the application communicates with data storage services, often called resource managers. The application performs its own specific operations needed to process the transaction. The application is responsible for maintaining the consistency of its objects. See Figure 1.

Because the KeyKOS microkernel supports cheap processes, KeyTXF uses several processes for each transaction, simplifying both system and application design. These processes are objects in KeyKOS terminology. Multiple instances of the application are used to allow more than one transaction to be processed at a time. Using multiple instances allows the application programmer to program with the simplicity of single threaded logic.

[1] The actual resolution depends on the hardware timing facilities available

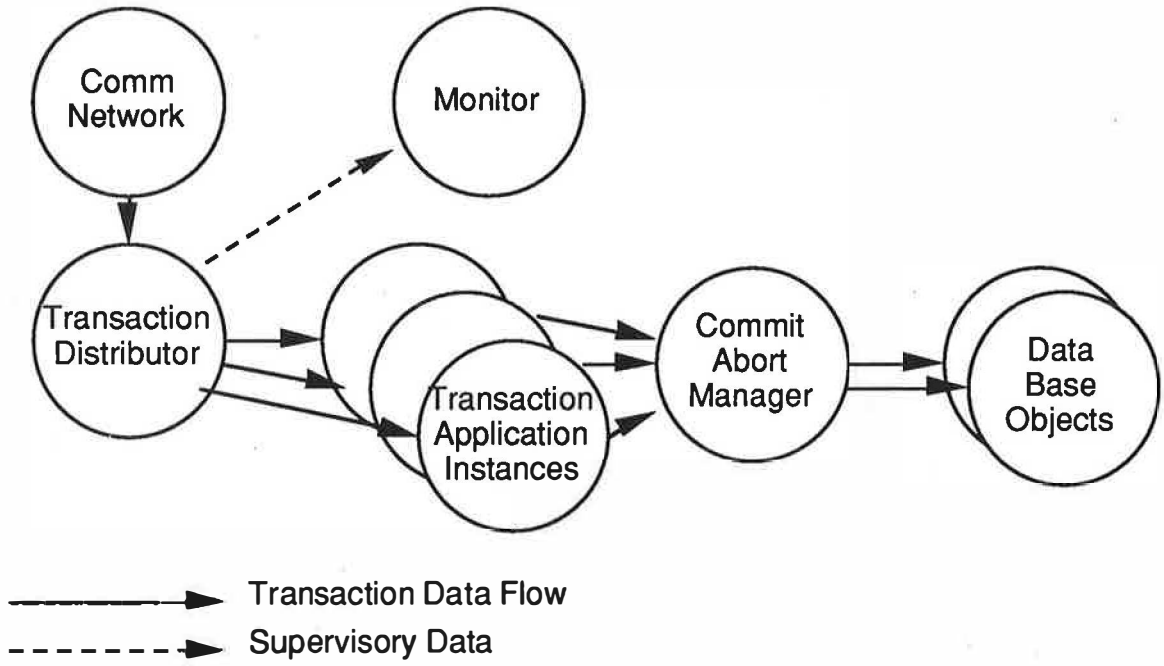


Figure 1: KeyTXF, Overall Structure

Both the Transaction Distributor and the Commit/Abort Manager dedicate an interface object to each application instance. The Transaction Distributor uses its interface object to send transactions to the application instance, monitor its status, and provide it with networking services. The Commit/Abort Manager uses its interface object, called the Commit/Abort Interface Object (CAIO) to save data to be committed as described below. KeyTXF uses these objects to serve many of the functions that the transaction ID serves in other systems.

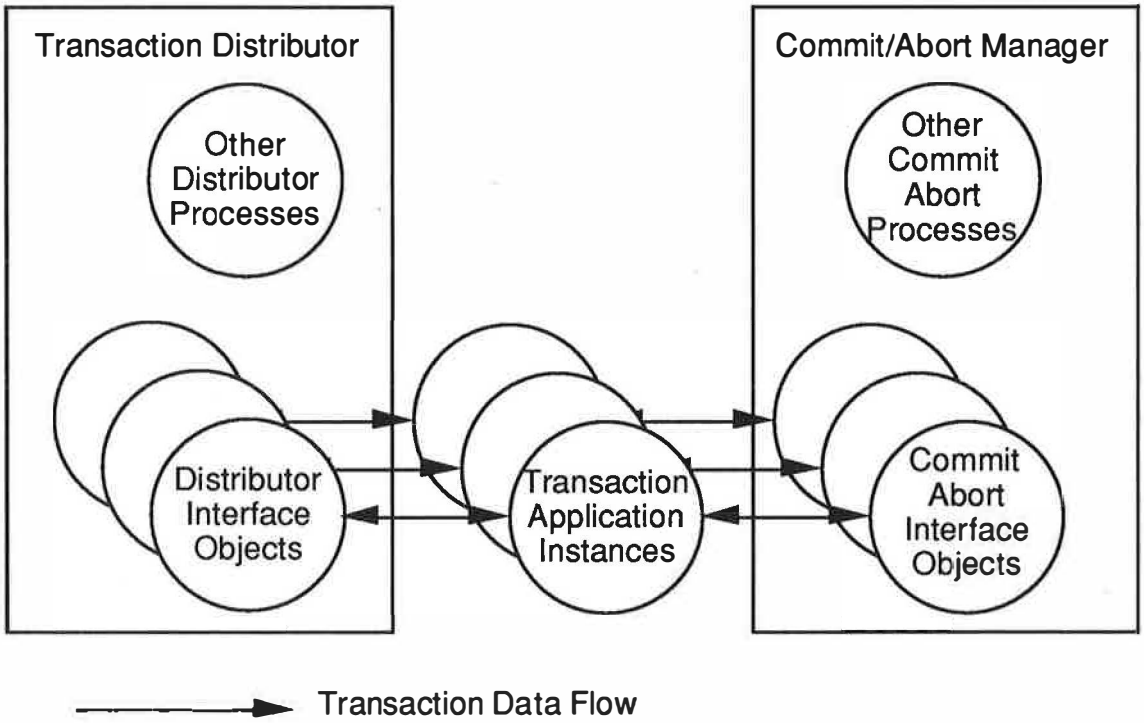


Figure 2: Transaction Distributor and Commit/Abort Manager Process Structure

4.1 Front end

Transactions enter the system as messages from a communications network. They are routed to the Transaction Distributor, which selects an available application instance to process the transaction and passes the message to the application. It returns messages from the application to the transaction source. It can also detect the need for restart recovery.

4.1.1 Transaction Distributor

In addition to selecting an available application instance to process the transaction, the Transaction Distributor is responsible for recovering from application program crashes by aborting any transaction that is in progress and notifying the transaction source that the transaction has failed.

4.1.2 Monitor

The Monitor provides an operator with a real time display of the status of the system.

4.2 Back end

The back end consists of the Commit/Abort Manager, and one or more database objects.

4.2.1 Commit/Abort Manager (CAM)

The heart of the system is the Commit/Abort Manager (CAM). The CAM ensures atomicity of transactions by supporting Commit and Abort operations. During transaction processing, it records all changes to the database objects. The actual database objects are not changed until the transaction is committed. When a commit is issued, the CAM writes all the database changes into a log, uses the Journalize Page operation to protect that log, sends the changes to the database objects, and then releases all the locks acquired by the transaction.

In addition, the CAM is responsible detecting whether the system has restarted, and recovering from the restart. During restart recovery, the CAM reads the log and re-sends the logged changes to the database objects.

The CAM is described in detail in section 5.

4.2.2 Database Objects

The database ensures isolation or serialization of transactions by means of record-level locking. The data storage objects do not need to know about commit/abort since that logic is in the Commit/Abort Manager.

The microkernel's object orientation helped us build a database out of smaller objects. We used a preexisting object type, called a record collection, which provided efficient indexed access to data. We used record collections to build a new object type, called a Multiple User Record Collection (MURC), which provides two enhancements to the basic record collection. The MURC allows multi-threaded access to data by spreading the data over multiple instances of the record collection (which is single threaded). In addition the MURC provides locking services. This construction resulted in a simple, efficient database, and took a new KeyKOS programmer two months to implement. Other data storage objects could also use wrappers to provide locking and/or interface translation.

The MURC goes through two stages of instantiation during system operation. The first stage occurs when a database is created. The MURC creates a number (specified by the caller) of record collections to hold the data. The caller has the option of arranging for these record collections to use space on different physical disk drives, to achieve parallel access. See figure 3.

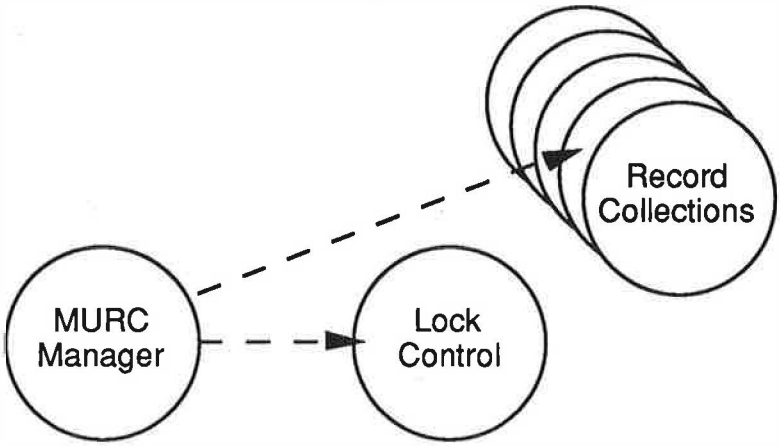


Figure 3: The MURC after a database has been created

The second stage occurs when application instances are created. Each application instance has an associated MURC Accessor (one for each database) to hold its locks and to select the record collection which holds the desired record. See figure 4. Figure 5 shows the how the CAM connects to the database objects.

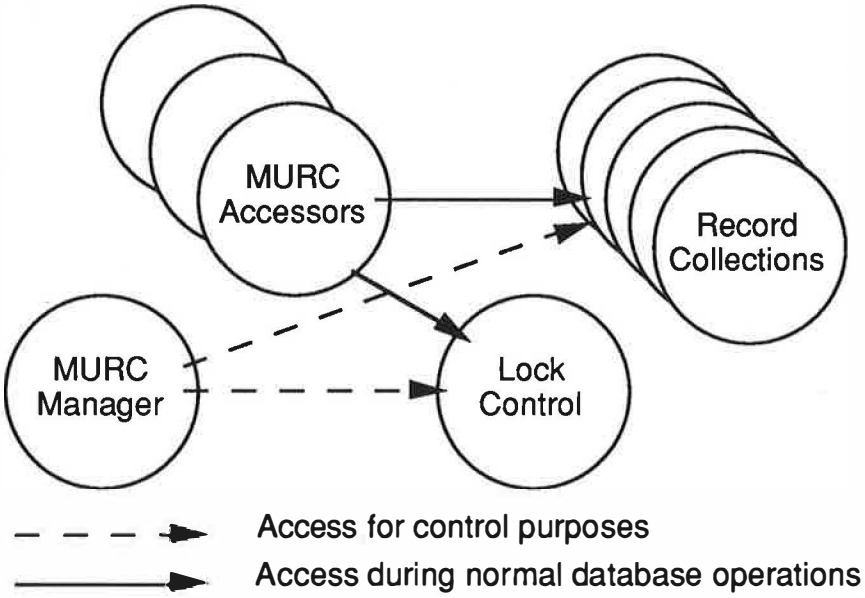


Figure 4: The MURC after three application instances have been created

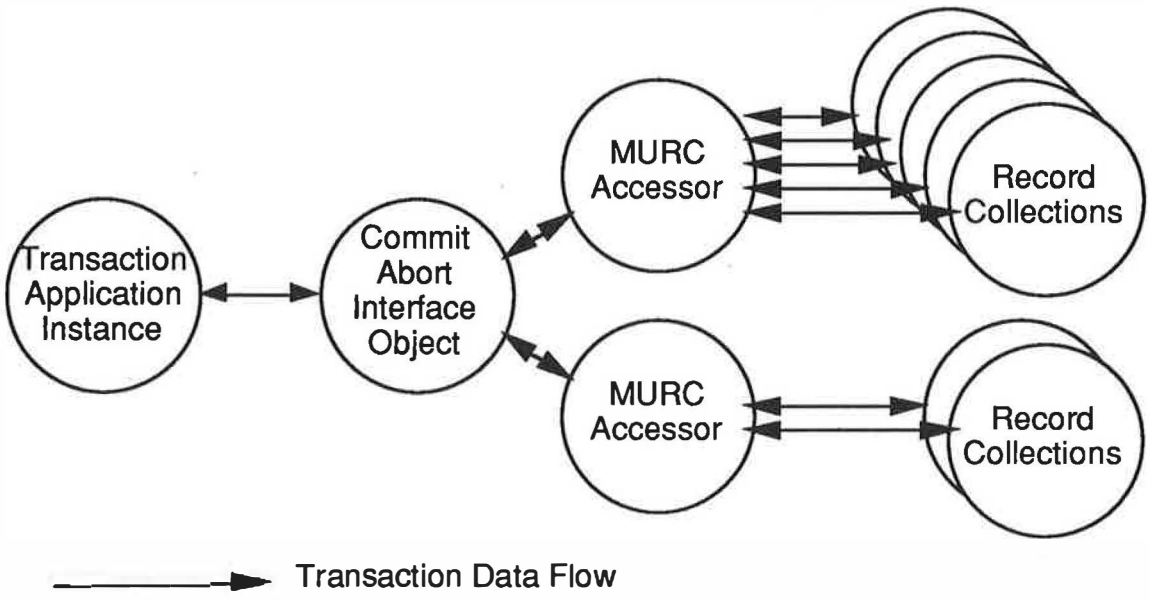


Figure 5: An application instance accessing two databases

5 Commit/Abort Manager Logic

5.1 The Log

The log contains information about all the transactions that have been committed since the last checkpoint. Each log entry consists of an entry header, and the data which defines the changes that were made to the database objects during the transaction, called the Local Event Queue (LEQ, see below). The entry header contains the length of the entry, and the internal application ID of the application instance responsible for this entry. The last log entry is followed by a header containing a length of zero which acts as an End of File (EOF) mark.

The log is maintained in a fixed set of pages used in a circular manner. Access to the log is serialized by a lock called the log lock. The CAM maintains a global pointer to the place for the next log entry. Each page of the log has a header and a trailer, which are completely independent of the entry headers. The log page header consists of a word which is used to ensure disk write consistency (described later), and the time stamp of the last log entry that was placed in the page. The trailer consists of another word for disk write consistency assurance. The pages of the log have time stamps that are monotonically increasing, up to and including the page containing the EOF mark. As a refinement, if the EOF mark would occupy the first location in a page, it is omitted; in this case, non-monotonically increasing time stamps will mark the EOF.

Since log entries that were made before the most recent checkpoint are not needed to recover the database, the log need contain no more data than is generated in a checkpoint interval. However, there are no significant costs (other than storage costs) in having the log larger than necessary. The CAM will force a checkpoint if the log space is exhausted. This action causes all transactions to wait until the checkpoint completes, and is undesirable because it introduces a noticeable delay in transaction processing. Allocating enough space for the log avoids this cause of delay.

5.2 Evolution of the Log

Understanding the logic of the Commit/Abort Manager requires understanding the states of the log. A somewhat simplified view of the log, ignoring transient states while the CAIO is writing to the log, starts with the end of the previous replay and proceeds to the start of the next replay.

At the end of replay, the Commit/Abort Manager initializes the Log Pointer to point to an EOF mark in the log. See figure 6.

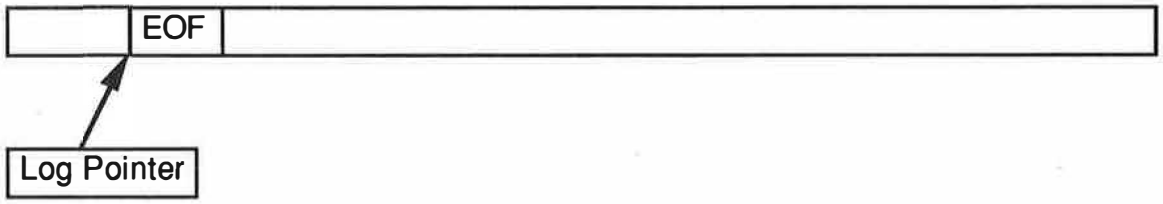


Figure 6: The state of the log immediately after replay

After transactions have been committed, the log contains some transactions that were committed before the most recent checkpoint, and some that were committed after it. See figure 7.

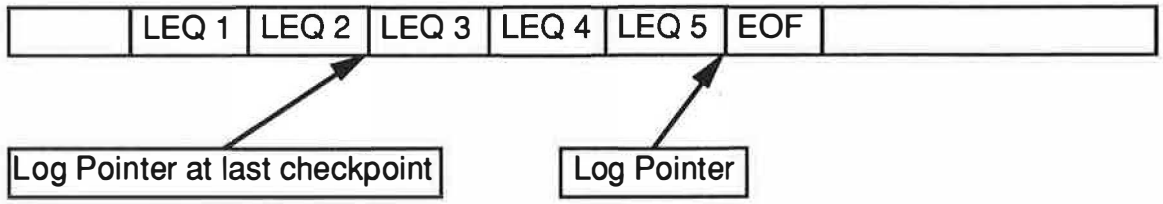


Figure 7: The state of the log after some transactions have been committed.

If the system fails at this point, the Log Pointer, the processes, and the databases (in fact, everything but the log), are backed up to their state at the last checkpoint. The log is not backed up because its pages are written out with the Journalize Page operation. The part of the log that must be replayed is the part between the log pointer and the EOF mark. See figure 8.

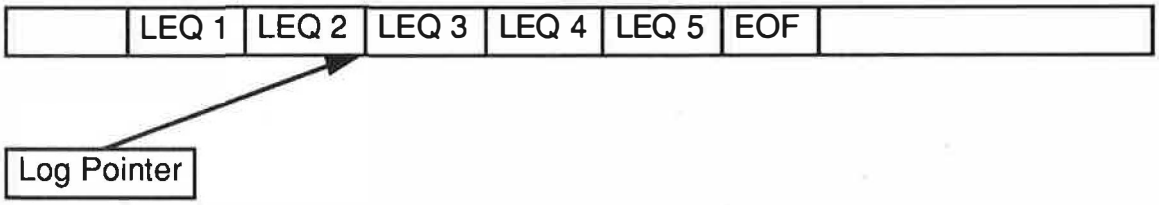


Figure 8: The state of the log after a restart has occurred and before replay.

5.3 Restart Detection

Most KeyKOS programs do not need to detect restarts. When KeyKOS restarts from a checkpoint, all their state: registers, memory, files etc. etc., are restored to the (consistent) state at the time of the checkpoint and the program runs from there, producing the same results. When a program uses the Journalize Page operation, it loses this consistency, and needs to be aware of restarts.

Consider the case of a CAIO about to store an EOF mark in the log. The simple-minded logic would be: (1) Write the LEQ into the log, (2) store an EOF mark in the log, and (3) call Journalize Page to write the log to disk. If a checkpoint is taken during step one, and the system continues to commit transactions, we end up with a log that looks like figure 7.

If the KeyKOS restarts from that checkpoint, then the CAIO, whose state was backed up to the time of the checkpoint, completes step one, which is not a problem since copying the LEQ is an idempotent operation. But it then stores an EOF in the log, truncating the log and "uncommitting" the transactions following it in the log. To avoid this problem, the CAIO checks for restart when storing EOF in the log and at other critical points, and executes special logic if it finds that a restart has occurred.

When the CAIO completes a replay, it saves the time of last restart in a global variable. To check for restart, it compares the time of last restart in the checkpoint page with the value it saved. If they are different, then a restart has occurred.

Any time the CAIO detects that a restart has occurred, it first aborts any transaction in progress. (If the transaction was committed before the restart, then its effects on the database will be restored by replay.) The CAIO then initiates replay of the log.

5.4 Commit/Abort Interface Object (CAIO) Logic

5.4.1 Begin Transaction Logic

A Begin Transaction operation is used by the CAIO only to ensure that the application is processing in the correct sequence. Because the CAIO's provide transaction protection at all times, KeyTXF always considers the application instance to be in a transaction. As explained below, KeyKOS does not use Transaction ID's, so when the application instance issues a Begin Transaction, there is no work to be done. This operation is not fundamental, and could be eliminated, saving a small amount of overhead.

5.4.2 During Transaction Logic

During transaction processing, the CAIO passes the read requests along to the database. When the application instance issues a read with lock, a write, or a delete, the CAIO saves data about the operation internally in a structure called the Local Event Queue (LEQ).

5.4.3 Abort Transaction Logic

An Abort Transaction operation causes the CAIO to discard the LEQ and release the database locks. Since no changes have been made to the database, that is the only processing necessary.

5.4.4 Commit Transaction Logic

The commit operation is performed by a single phase commit. A simplified view of what happens when the application issues a "commit transaction" operation, is that the CAIO obtains the log lock, appends the LEQ to the log, releases the log lock, sends the changes recorded in the LEQ to the database, signals the database to release the locks, and returns "commit successful" to the application instance.

Several optimizations contribute to the subtlety of the actual logic. Journalizing the pages of the log is done by independent threads for parallelism. The sequence of updating the database and journalizing the pages is arranged to maximize the probability of committing several transactions with a single journal write. And, as mentioned above, special logic is required to preserve the log when a restart occurs.

As the Commit/Abort Manager appends to the log, it needs to advance through the pages of the log. The procedure *getnextlogpage* is used to advance to the next page, ensuring that committed transaction entries are not nullified. The logic of *getnextlogpage* is:

Save the current time (called the new time stamp). If all goes well, this will be used as the new time stamp in the header of the current log page.

Check for restart. By ensuring, after storing the time stamp, that a restart has not occurred, we ensure that the time stamp stored is earlier than any need for replay. If *getnextlogpage* detects a restart, it does not update the time stamp in the log page header, and a replay will be started.

Copy the old time stamp in the header of the current page. Check that this time stamp is earlier than the new one. (It could be later if a checkpoint occurred after the check for restart, subsequent logging had written later entries into the page, and the system then restarted from that checkpoint.) If it is earlier, then use Compare and Swap to update the log page header time stamp passing the old time stamp as the "old" value and the new time stamp as the "new" value. This update will only succeed if the value in the page has not changed. If the value has changed, a restart has occurred.

Get the capability to the next log page and make the page addressable. If the new page header's time stamp is more recent than the last checkpoint, then the log has wrapped. Take a checkpoint to allow the log page to be reused.

Fork a log write thread to journalize the just filled page.

The complete logic to commit a transaction is:

Obtain the log lock.

Check for restart.

Write a log entry header. Writing the header destroys the EOF mark. The end of the log must be detected by time stamps until the EOF mark is written at the end of the new entry. Call *getnextlogpage* whenever a new page is needed to write the log entry.

Copy data from the LEQ to the log using *getnextlogpage* as needed.

If there is room in the current page, then write an EOF mark (a length word of zero), using the following logic to avoid truncating a log after a restart:

Store the current time stamp.

Copy the old time stamp in the log page header, and the current value of the word where we will write the EOF, into local variables (subject to checkpoint).

Check for restart. If the time stamp stored is earlier than or the same as the value found in the log page header, then a restart has occurred.

Use Compare and Swap to update the EOF mark using the value saved above as the "old" value. If the compare part of the Compare and Swap failed, a restart has occurred.

Use Compare and Swap to update the time stamp in the page header using the value saved above as the "old" value. If the compare matched then the EOF mark has been successfully stored.

Otherwise, the time stamp is different from the one saved and the system has restarted. If the time stamp we were trying to store is the same as the one now in the log page header, then the log entry being made is the last one in the log and the EOF mark just stored should remain in the log. Otherwise additional entries have been made in the log after ours and if the EOF mark is still zero, we must restore the old value (known because of the Compare and Swap). Continue with the normal restart logic.

Update the log pointer to point to the next log slot.

Release the log lock.

Perform a pre-page operation on the next few log pages to avoid a paging operation while the log lock is held.

Save the time stamp from the last log page for this log entry for use in checking for completion of the Journalize Page operations.

Process the entries in the LEQ in the order they were made. The lock entries need no action because the locks have already been obtained. For write or delete entries, call the appropriate database instance to perform the write or delete. Note whether there are any unlocked append entries, but do not perform them.

Release all database locks.

If there were any unlocked append entries, reprocess the LEQ to perform them. This logic allows an application to maintain a sequential log of transactions without that log file forcing serialization.

Ensure that the entire log entry has been journalized by checking that log writes for all log pages older than or the same age as the last page of the entry have completed. If it is necessary to schedule a write, this transaction may be delayed up to 100ms to allow other transactions to be committed by the same write. Journalizing a log page is done by forking a thread. Releasing the log lock before performing the Journalize Page operation allows one disk write of the log to commit more than one transaction if their log entries fit in the same page.

Return "transaction committed" to the caller.

5.5 Replay

During replay, pages of the log are processed sequentially beginning from the log pointer. If a log page is found with a time stamp earlier than that of the previous page, then the current log entry is incomplete, it is discarded, and the log is considered to be at EOF.

Replay runs in a process of its own, but calls the CAIO's associated with each application instance to actually perform the database updates. Replay first gets the log lock to ensure all CAIO's writing into the log have completed their writes. Replay then processes the log, starting from the log pointer saved by the checkpoint, and continuing until it detects log EOF. It releases the log lock before calling the CAIO's, to ensure that they can complete their attempts to log a transaction (by aborting it) and become available to help in the replay. They regain the locks and perform the database updates. When replay is completed the application and CAIO's are destroyed and rebuilt to avoid application errors that could continue a transaction that was entered before the restart.

6 Disk Failure Protection

One non-obvious area of failure is when the computer system fails when halfway through writing a page. This failure can occur if the disk sectors are smaller than a page and only some of them have been written. If this occurs while the Journalize Page operation is writing a page, it is possible that, after restart, the page will have data from two separate times. The microkernel provides support for recovery from failures of this kind.

The KeyKOS microkernel protects against this occurrence by allocating log pages in mirrored disk storage, and completing the write operation on one page before starting it on the other. The Journalize Page operation allows pages to be placed in a special state, where the kernel will use the first and last word in the page to hold a serial number of the write count. If a page in this state is read and the serial number at the start of the page does not match the serial number at the end, then the page is known to be corrupted and the other copy is read.

This technique of failure protection can be defeated by certain algorithms in the disk controller. If for example, the disk controller caches sectors of a page and writes the first and last sectors before writing one of the middle ones, a failure may escape detection. Most SCSI disk systems do not specify whether they use such algorithms.

7 Comparison with Other Systems

It is instructive to compare the structure of KeyTXF with that of other systems. The KeyTXF structure takes advantage of the KeyKOS microkernel's object orientation and efficient support for many objects. Without these kernel features, the design approach used in KeyTXF would not be fast enough to be useful.

Other systems generate a unique Transaction ID for each transaction that is processed [3]; KeyTXF does not use Transaction ID's at all. A transaction is identified with the application instance that generates it. To distinguish transactions from different application instances, an application ID is used in the log. The application ID is reused for each transaction from the same application instance.

The Commit/Abort Interface Object serves to carry the identification of the application ID, and therefore the transaction, to the CAM. It is worthwhile to note the generality of the object-oriented microkernel paradigm here. Systems that use Transaction ID's must carry that ID in messages to entities that are involved in the transaction, such as databases and the transaction processing monitor. Often the Transaction ID is passed implicitly using an operating system level mechanism that is specific to transaction processing.

The KeyKOS microkernel was not designed specifically for transaction processing and has no such specific mechanism. But the object-oriented design of KeyTXF makes such a mechanism unnecessary. Lacking such a mechanism, the microkernel remains small and fast.

The CAIO's are the key to this design. For each application instance, the CAIO mediates the application's communication with the database objects. The interface objects are part of the CAM; they are responsible for identifying the different application instances and therefore the transactions. The application code does not need to concern itself with identifying transactions.

Mediating the application's access to the database objects is possible because of the security features of KeyKOS. An application instance has access to the CAM, but does not have any direct access to the database objects. Only the CAM has access to the database objects. Thus the CAM is in a position to control the application's access to the database objects by interposing an interface object. The KeyKOS microkernel allows this type of access control to be strictly enforced.

It is worth pointing out that an application instance is a general KeyKOS object, and as such is not restricted to a single process. It can create processes and objects, and can communicate with existing objects. The interface objects ensure that a database object is transaction protected even though the application instance may pass its capability to the CAIO on to other objects.

8 Experience

8.1 Benchmark Results

An implementation of the TP1 performance benchmark[1] in 1986, showed that KeyTXF can process 18 transactions/second on an IBM 4341 (a 1.3 MIPS processor) with two channels of 1.5 megabyte/second disk. This is equivalent to 13 transactions/second/MIPS. There were 3.3 disk I/O's, 3 X.25 packets and 76,000 instructions per transaction. Additional benchmarks show that performance scales linearly with CPU speed if the I/O system is also upgraded for increased capacity. This compared with 0.7 to 7 transactions/second/MIPS for a variety of other systems at the time [7]. IBM's TPF achieved about 22 transactions/second/MIPS, but offers no protection; in TPF, all applications run in supervisor state.

The superior performance resulted from the microkernel design and implementation. The microkernel's checkpoint mechanism supports the caching in virtual memory of information normally stored in disk files, while maintaining consistency and durability.

8.2 Banking application

A commercial bank designed and implemented an application to process credit card authorization transactions using KeyKOS and KeyTXF. Though no formal measures were made, the development required approximately half the effort that it was estimated would have been required using traditional transaction processing systems.

9 Conclusions

The KeyTXF experience shows that it is possible to gain the advantages of microkernel architecture, object oriented operating systems, and high performance transaction processing in one system. Object orientation made it possible to keep the design of the transaction processing system and its applications simple, even to the extent of eliminating the use of Transaction ID's. The microkernel architecture allowed us to achieve superior performance. We believe that reliability assertions require that disk system manufacturers provide sufficient specifications to allow software recovery from system failures that occur while writing.

10 Acknowledgements

KeyTXF was developed by William Frantz at Key Logic in 1986. The MURC was written by Susan Rajunas. This paper is dedicated to the memory of Codie Wells.

11 References

[1] Anon, et al., "A Measure of Transaction Processing Power", Datamation, April 1, 1985, pp 112-118

[2] Bomberger, Alan et al., "The KeyKOS® Nanokernel Architecture", Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures, USENIX Association, April 1992, pp 95-112

[3] Gray, Jim, and Reuter, Andreas, *Transaction Processing: Concepts and Techniques,* Morgan Kaufmann, 1993

[4] Hardy, Norman, "The KeyKOS architecture", Operating Systems Review, v.19 n.4, October 1985. pp 8-25

[5] Landau, Charles R., "The Checkpoint Mechanism in KeyKOS", Proceedings of the Second International Workshop on Object Orientation in Operating Systems, IEEE, September 1992, pp 86-91

[6] Levy, Henry M., *Capability-Based Computer Systems,* Digital Press, 1984.

[7] Published industry figures.

[8] Schmuck, F., and Wyllie, J., "Experience with Transactions in QuickSilver", Proceedings of the 13th ACM Symposium on Operating Systems Principles, 1991, pp 239-253

[9] Thatte, Satish M., "PERSISTENT MEMORY: A Storage Architecture for Object-Oriented Database Systems", Proceedings of the 1986 International Workshop on Object-Oriented Database Systems, IEEE and ACM, 1986, pp 148-159

From V to Vanguard: The Evolution of a Distributed, Object-Oriented Microkernel Interface

Ross S. Finlayson, Mark D. Hennecke, Steven L. Goldberg*
Apple Computer, Inc.

Abstract

The Vanguard operating system kernel was designed and implemented as a research testbed for distributed applications and higher-level operating system services. Using the design of the V-System as a starting point, we developed an extensible set of operating system services, organized in an object type hierarchy. We also implemented a modular OS (micro)kernel that implements these services.

An important part of any microkernel design is its exported interface, as the design of this interface affects the ease with which programmers can develop higher-level operating system services on top of the kernel. In this paper we describe several notable features of the Vanguard microkernel interface—in particular, its process and object model, its object identification scheme, and its use of group communication. We show how these features lead to a simple yet powerful interface that avoids the need to provide an excessive number of operations.

1. Introduction

An important part of any microkernel design is the interface that the microkernel exports to higher-level OS services. In fact, this interface can be said to define the microkernel itself. A microkernel interface must be sufficiently rich to easily support all possible higher-level services that could be built on top of it, including value-added OS services (such as file systems), application program interfaces and runtime libraries, and emulation of other OS interfaces. At the same time, the interface should not hinder programmers by being too complex or inconsistent. Fortunately, compared to application programming interfaces, microkernel interfaces are usually less constrained by backwards compatibility concerns, and this gives microkernel architects a valuable opportunity to produce well-designed interfaces.

Modern operating system interfaces are frequently based upon the object model [7], which is well suited for kernelized OSs. Because the implementation of a kernel service is hidden from its interface, clients of the service are oblivious to whether the service's implementation exists inside or outside the kernel (or even on a remote node). Inheritance and polymorphism further simplify the interface (by reducing the total number of operations), and allow higher-level OS software to seamlessly extend the interface, if desired.

In this paper we describe several key features of the (object-oriented) interface to the Vanguard operating system (micro)kernel. Vanguard [5] was designed as a research testbed for distributed applications and higher-level operating system services, and has been implemented both on raw hardware (Motorola 88100-based coprocessor boards) and hosted on top of other operating systems (Macintosh OS and Unix). Vanguard's OS interface is intended to be used by one or more levels of system software above the OS kernel, although not necessarily by application code, which may wish to use existing APIs.

Vanguard's design was heavily influenced by that of the V-System [3]; this paper concentrates on those features of Vanguard that either did not exist in the V-System, or are extensions of similar features in V.

2. Processes, Invocation, and Objects

Like the V-System, Vanguard is defined by a suite of request and response messages. The units of control structuring and concurrency are lightweight threads (called *processes*), which communicate using a synchronous "Send-Receive-Reply" IPC model.

In V the object of a "Send" operation was a server process id. To identify a particular entity or service (e.g., a window or an open file) that is implemented by this process, request messages would typically contain a separate local id, interpreted by the server, that identifies this entity. This style of request message—denoting a distinct local object managed by the server process—was so common in V that in Vanguard we chose to make it a fundamental part of the IPC model. In Vanguard the object of each "Send" invocation is a (128-bit) *object id*. Although client processes treat object ids as opaque, they really consist of two 64-bit portions: a server *process id*, and a server-relative *local id*. The kernel uses the process id to route each message, delivering it to the designated process if it is local, otherwise delivering it to a separate

* Author's current affiliation: SunSoft, Inc. (finlayson@eng.sun.com)

transport server (similar to Mach's "netserver" [1]). The transport server then delivers the message to the appropriate remote kernel, using a transport protocol suited to the interconnect. (On a local area network we use the same VMTP protocol [2] used by the V-System.) The local id portion of the id is not used for message routing—instead, the allocation, interpretation and use of the local id is the responsibility of the server processes.

An object id is valid as long as the server process designated by its *process id* remains running. A rebooting kernel does not attempt to resurrect servers with their old ids. This policy is appropriate for a microkernel object system, as opposed to a more application-oriented object system such as CORBA [8], where object references can persist across server shutdowns. In Vanguard, longer-lived references, when desired, can be created using a higher-level naming mechanism (described below).

Object ids identify all operating system services, including those services that are exported by the kernel itself—for example devices and address spaces. Processes are also objects, managed by a kernel *process server*. (Thus, each process server is conceptually managed by itself, which terminates the recursion.) As an object, a process can be operated on just like any other object, by sending a message to its manager, the process server. For example, a file *f* managed by server *s* can be deleted by sending a "delete" message to process *s*, invoking the object (i.e., file) whose id is *(s, f)*. In the same way, the process *s* can be destroyed by sending a "delete" message to the process server *p*, invoking the object (i.e., process) whose id is *(p, s)*.

3. The Object Type Hierarchy

To reduce the complexity of the Vanguard interface, objects are classified in an abstract type hierarchy. Each type defines a set of operations (with corresponding opcodes, request messages and response messages) that are applicable to objects of this type, and inherited by subtypes. For example, the "delete" operation described earlier is one of the operations defined on the base type, "Vanguard Object". (The V-System's interface, in contrast, had separate "delete" messages for each kind of object.)

Subtrees of the type hierarchy include "I/O objects" (files, disks, consoles, networks and other devices) and "character-string-named objects" (described below).

The Vanguard interface, being message-based, is programming language independent—there is no fundamental relationship between Vanguard objects and the lighter-weight objects defined by object-oriented programming languages. This allows clients and servers to be written in different programming languages, including non-object-oriented languages, as long as they conform to the Vanguard protocols. However, if clients are programmed in an object-oriented language, then it is natural to use programming language objects as message stubs encapsulating Vanguard objects. Similarly, programming language objects can be used in server implementations. At present, these client and server stubs are written by hand, but in principle they could also be generated automatically from specifications written in a common "interface description language", as is done in many other systems. Vanguard currently has client and server bindings for C++, and client bindings for Common Lisp. (Additional details of Vanguard's C++ language binding can be found in [6].)

4. Object Groups

One of the more noteworthy features of the V-System was its notion of a *process group*. A process id could be used as a group id, representing an arbitrary number of processes. A message sent to a process group would be delivered (1-reliably) to all members of the group. The process group mechanism takes advantage of multicast, if this is supported by the underlying network or interconnect. A process groups can be used in two possible ways: for "resource location" or "multi-destination delivery". In the first case, the group id provides a level of indirection: a new process can join an already-known group, and can be reached by sending to the group id. In the second case, the group id represents a group of several processes that can be notified collectively using a single message.

Vanguard extends this concept to that of an *object group*. An object group id is identical to a regular object id, except that either the "process id" portion *or* the "local id" portion can denote multiple members. Thus, an object group can be managed by an arbitrary number of server processes, each of which may in turn implement an arbitrary number of local objects.

(The "process server" described earlier is an example of an object group, because there are really several individual process servers, one for each kernel in the system.)

When an object group is invoked, the invocation's request message is delivered (1-reliably) to each server process, which in turn delivers the message to each local object (if any). For each delivery, a server process can choose either to return a response, or not to respond (or more precisely, to *discard* the request). (Multiple responses from the same server are aggregated into a single response message.) The client process can also specify how many responses it desires. The default value is 1, meaning that the client remains blocked until the first response returns, or until the kernel(s) can determine that no responses will

return. A value of 0 indicates that the invocation is a "best efforts" datagram send, with no guarantee of delivery.

Object group membership is completely decentralized, and group "join" and "leave" operations are handled efficiently. In particular, a request to join object *o* to a group is sent directly to *o*'s kernel, and a network multicast occurs only if the group was previously unknown to this kernel.

Object groups provide a convenient way of operating on collections of objects, without having to complicate the interface by introducing an additional set of opcodes and message formats solely for this purpose. For example, each directory in Vanguard's character-string name space (see below) has an associated object group representing the members of the directory. Any operation that can be applied to any individual member of the directory can also be applied to *all* members of the directory, simply by invoking its object group instead.

5. Decentralized Character-String Naming

Along with a low-level identification mechanism (such as the object id scheme described earlier), it is also useful for an operating system to have an additional, higher-level naming mechanism that allows more permanent, human-understandable names to be assigned to certain system objects. This kind of naming mechanism is typically provided by the file system—i.e., above the kernel level in most microkernel-based designs. However, such a naming mechanism, if efficiently implemented, can also be usefully applied inside the kernel. In particular, the ability to give important kernel services (such as devices) character string names can make the kernel easier to configure, maintain and debug.

Vanguard uses the same naming mechanism as the V-System: *decentralized naming* [4]. To review: The main idea of decentralized naming is that the system has no (physically or logically) centralized name servers. Instead, each server that wishes to enter an object into the name space must also be prepared to handle "lookup" operations on this object. A directory in the name space may be distributed; a name in such a directory is resolved by multicasting a "lookup" operation to all servers that implement parts of this directory—i.e., by invoking the object group for the directory. Only the server that actually implements this name will return a response; the rest will simply discard the request.

To reduce the number of multicast messages, the V-System allowed clients to maintain a cache of mappings from hierarchical name prefixes to server pids, and would consult this cache before resorting to multicast. Vanguard's implementation of decentralized naming does not currently include client caching, but it could be implemented by adding it to the client language stub for the "lookup" operation—in effect making this stub a "proxy" [9] for the actual "lookup" request message.

Decentralized naming allows us to assign character-string names to several of Vanguard's exported kernel objects with little overhead. In particular, there is no separate "name server" process, either inside or outside the kernel.

6. Request Chaining, and High-level Ids

Quite frequently, the result of a Vanguard object invocation is also a Vanguard object—perhaps even the same object as the original. Thus, Vanguard response messages contain an optional *result object id*; a client may then use this id as the target of a subsequent invocation.

Unfortunately, the accumulated round-trip delays from a sequence of object invocations may be costly, especially over a network. To alleviate this problem, Vanguard's protocols allow a sequence of object invocations to be chained into a single message, in the common case where each invocation in the sequence is applied to the object resulting from the previous invocation. For example, to delete the object whose character string name is "foo" (in directory <dirId>), one could combine the two requests "lookup *foo* " and "delete" into a single message, and use this chained message to invoke the object <dirId>. A response message would be returned only from the last subrequest ("delete"), and not from the intermediate "lookup". Chaining not only gives a performance benefit, but also simplifies the OS interface by allowing new operations to be created by combining existing operations, rather than inventing (for example) a separate "delete by name" operation.

One could imagine extending this idea even further, by allowing requests to be combined in more complex ways—for example by introducing variables, conditional expressions and loops. Such extensions, however, would considerably complicate the implementation of servers. Therefore we have chosen to support only linear chains; these are simple to implement, and are especially useful.

An interesting consequence of the chaining mechanism is that a prefix of a chain can be used as an alternative, *high-level* form of object id. For instance, from the example above, the object id <dirId> and the request "lookup *foo* " can be encapsulated together as an opaque, high-level id. This object id (like any other) can be invoked with a request message; this request message will be chained onto the end of the "lookup" request. Thus, the "lookup" operation will be evaluated anew on each invocation.

A programming language stub for an object encapsulates an object id—either a simple, low-level object id, or a high-level id. Client code does not know or care either way. High-level ids are used quite frequently in Vanguard, for example in the virtual memory system, where a high-level id may (transparently) represent a subrange of an address space, rather than a whole address space.

7. Summary

We have presented a summary of the interface exported by the Vanguard microkernel, showing in particular how a rich operating system interface can be built from a relatively small set of basic operations. We have described three separate techniques that make this possible. First, objects are organized in an inheritance hierarchy. Second, the interface supports object group ids, which appear to clients exactly like regular object ids. This makes it possible to operate on groups of related objects without introducing new operations. Third, chains of dependent operations can be grouped together into single requests. Prefixes of these chains can also be used as alternative, higher-level ids. We believe that these techniques have widespread applicability to many microkernel designs.

8. References

1. Accetta, M. J., R. V. Baron, W. Bolosky, D. B. Golub, R. F. Rashid, A. Tevanian Jr. and M. W. Young. Mach: A New Kernel Foundation for Unix Development. Proceedings of the Summer USENIX Conference. 93-113, 1986.

2. Cheriton, D. R. VMTP: A transport protocol for the next generation of communication systems. SIGCOMM '86 Symposium: Communication Architectures and Protocols. 406-415, 1986.

3. Cheriton, D. R. The V distributed system. Communications of the ACM. 31(3): 314-333, 1988.

4. Cheriton, D. R. and T. P. Mann. Decentralizing a global naming service for improved performance and fault-tolerance. ACM Transactions on Computer Systems. 7(2): 147-183, 1987.

5. Finlayson, R. S., M. D. Hennecke and S. L. Goldberg. Vanguard: A protocol suite and OS kernel for distributed object-oriented environments. IEEE Workshop on Experimental Distributed Systems. 1990.

6. Finlayson, R. S., M. D. Hennecke, S. L. Goldberg, J. L. Coolidge, A. G. Parghi and E. W. Sznyter. Object-Oriented Communication and Structuring in Vanguard. IEEE International Workshop on Object Orientation in Operating Systems. 112-113, 1991.

7. Jones, A. "The object model: A conceptual tool for structuring software." Operating systems: An advanced course. Bayer, Graham and Seegmueller ed. 1979 Springer Verlag.

8. Object Management Group, Inc. The Common Object Request Broker: Architecture and Specification. 1992.

9. Shapiro, M. Structure and encapsulation in distributed systems: The proxy principle. Sixth International Conference on Distributed Computer Systems. 1986.

Design and Implementation of an Object-Orientated 64-bit Single Address Space Microkernel

Kevin Murray, Tim Wilkinson, Peter Osmon – SARC, City University
Ashley Saulsbury – Swedish Institute of Computer Science
Tom Stiemerling, Paul Kelly – Imperial College

Abstract

In the mid eighties, the System Architecture Research Centre at City University developed a message-passing, UNIX compliant micro kernel (*Meshix*) for our own scalable distributed memory architecture (*Topsy*). Over the last two years we have been engaged in a research programme aimed at learning from this experience, and developing a new operating system based on these lessons. The result is the *Angel* microkernel. This paper sets out the lessons we have learnt from *Meshix*, how this has influenced the design of *Angel* and outlines our current design of *Angel* and its C++ implementation. We will also describe our future plans and hopes for *Angel*, and the lessons that we have learnt from the design and implementation process.

1 Introduction

Almost all modern operating systems are being designed using microkernels [1, 2]. The microkernel architecture can be said to encompass good software engineering practice: small code "units" that are insulated from each other together with a minimal amount of critical code (the microkernel). They also introduce an "open system architecture" due to the ease with which additional services can be provided and used.

Virtually without exception, however, microkernel architectures use message passing as the basis of communication, implementing the client-server paradigm upon this using remote procedure call techniques (RPC). Message passing offers an apparently ideal structuring mechanism — it isolates one "unit" from another, requires only a minimal microkernel (message passing and process control), and allows extra services to be provided simply by registering the service which then receives and processes the messages.

Meshix is typical of such microkernel based, message passing operating systems and was developed several years ago [3]. Over the last few years we have been looking at its structure and performance in a very critical manner to decide how to improve upon it — in essence we are trying to evaluate whether or not the message passing microkernel is as good as it seems. This has shown there are a number of issues that have not yet been addressed by most current message passing microkernel architectures, or which have only been addressed with limited success or requiring complex restructuring of the system. It is to tackle these issues that *Angel* has been designed.

This paper will outline the issues behind the the design of *Angel*, in addition to the actual design and implementation of *Angel*. Although *Angel* moves away from a message passing structure, we will show how its most important use — RPC — can be very efficiently implemented in *Angel* using LRPC techniques, and how it maintains the essential isolation of the systems into protected "units". We will also mention some of the other work that we are applying to *Angel*, principally in the areas of fault tolerance and scalable I/O systems.

2 Shortcomings of Meshix

The original goal of *Meshix* and the *Topsy* architecture was to produce a scalable, parallel multiprocessor[1]. To help achieve this goal a dedicated point to point network with custom, virtual cut-through routing chips were developed which supported a raw bandwidth of 10 Mbytes/sec. To a reasonable extent the scalability goal has been achieved. Unfortunately its communications performance is only about 100 K/sec, as seen by a user process, and there is limited support for parallel programming. The reason is largely to do with two factors: the nature of microkernels, and the adoption of UNIX. The adoption of System Vr3 UNIX as the primary interface to *Meshix* means there is no support for parallel programs, forces the use of UNIX heavyweight processes and limits the IPC mechanisms.

2.1 Microkernel

In a microkernel architecture, there is an inherent performance loss caused by information exchange between services and clients. Typically a client collects the information it needs in its own private address space, independently of the server. When it wishes to exchange information with a server (probably to obtain some service), it first creates a message containing this information and then requests the microkernel to convey this to the server. Usually this involves several context switches and some data copying, remapping or cross-machine transferral, all of which are known to be costly actions.

The Chorus group [4] (among others), has done much work to overcome this. The methods used include replacing context-dependent addresses with unique addresses, so speeding up message delivery whilst reducing security, combining mutually trusted servers into a single address space (and hence protection domain), so reducing context switches, and by placing all of the IPC management into the microkernel. In addition they use the lightweight RPC optimisation developed for the DEC Firefly system [5] to improve the speed of RPCs. All of these modifications have required non-trivial alterations to the operating system's structure and increased the complexity of the system. It is our belief that communication (or more generally co-operation), despite the above optimisations, is still slower than desirable and more complex an operation than need be.

We performed a set of detailed measurements of the speed of the *Meshix* communication system [6, 7] to help identify the causes of communications costs, to better understand these costs, to find ways to reduce or eliminate them and to help develop a simpler mechanism. This study concluded that many, though not all, of the costs are an inherent fact of using message passing in multiple protection domains and the numerous context switches and data copying or remapping that this caused. During this study, it also became apparent that much, often unmeasured, time was spent in *preparing* the data for transfer.

As a comparison, we modelled the behaviour of a very simple distributed shared memory (DSM [8]) scheme with an amount of hardware assistance comparable with the current *Meshix* message passing system. The conclusion was that this would easily outperform the current message passing system, used in *Meshix*. This lead us to believe that the shared memory paradigm should be at the base of future parallel operating systems, replacing the message passing that is currently in use. This is in agreement with several other researchers and manufacturers [9, 10, 11].

2.2 UNIX

The UNIX process model provides every process with the illusion of a complete computer for itself. But whenever the process tries to access anything other than the processor or the data currently residing in memory, it may suffer a context switch as another process is given the chance to run. A context switch involves the exchange of a large amount of information beyond the processor's context including the memory map and extensive operating system information. This is called a heavyweight process model.

1. It should be noted that this is a somewhat different goal from some other systems which seek to produce a distributed computing systems.

Since *Meshix* provides a UNIX programming model, the process model it implements is that of UNIX: distinct heavyweight processes. The heavyweight process model is costly [12] due to its extensive amounts of state, and this reduces the benefits of writing programs in parallel, although this may be overcome to some extent using threads packages. Unfortunately, these threads are not real first class objects in the operating system and certain system operations for one thread affect others (eg. blocking). Even then it is still nearly impossible to share these threads between processes, unlike such systems as Psyche [13], to achieve the required flexibility (such as cost effective load balancing). For efficient parallel programming a lightweight, flexible and extensible process model is needed. This is one where changing from one thread to another is exceptionally cheap, where the actions of one thread do not necessarily affect another and where exchanging processes should also be cheap.

2.3 Support for Parallel Programs

Meshix provides no synchronisation primitives other than those implicit in message passing. Any others are built using messages. This means that if co-operating processes need to use synchronisation other than messaging, it is slow and limited by the characteristics of the messaging system. In a scalable parallel machine, real parallel programs will require efficient and varied synchronisation mechanisms (e.g. barrier synchronisation) and better support for them must be provided.

Additionally, much work has been done on load balancing in many systems and support for this is important to parallel programs since it is necessary for them to distribute their work over the machine efficiently. With a general purpose computer, the load on various parts of the system can change, and to maintain the efficiency of the applications running on such a system, it is necessary to re-allocate work between available processors. This is at the heart of load balancing, and naturally in a scalable parallel system it is important that, at the very least, there is support to allow this to be done.

3 The Angel Design and Single Address Space Architectures

From our experience with *Meshix*, and as a result of our studies both of *Meshix* and other systems, it was decided that *Angel* should have the following characteristics:

- It should not support message passing, but use shared memory to support a single address space. This decision was taken to tackle two problems: the lack of speed of the message passing model, as outlined above, and to improve the context switch time by removing the need to flush various caches, which has been noted as the most costly part of the context switch operation.
- It should provide a protection mechanism which is not part of the process. This decision was taken to allow a far greater flexibility in protection scheme, and allow more than one process to operate within the same domain to increase speed when necessary. It is also a logical step following the above point in which we had divested address translation from the process.
- It should allow processes to be informed of the actions of the operating system on their behalf. This decision is aimed at allowing threads within a process to become first class citizens of the operating system and to allow the process to partake in scheduling decisions that may affect it.
- It should use a minimal microkernel. None of our studies of *Meshix* showed a flaw in the microkernel design; in fact many of our experiences with *Meshix* have shown how vital the microkernel design is. The problems identified have been tackled by the above alterations to the architecture, so *Angel* remains a microkernel. However, as the implementation section will show, there is even less in the *Angel* kernel than in many other microkernels.

The following sections will outline the main characteristics of the *Angel* design.

3.1 SASA

Most importantly, *Angel* is a Single Address Space Architecture (SASA), like such systems as Multics [14], Psyche [15], and Opal [16]. A SASA is one in which there is only one address space shared by

the entire system (all the processes, servers and the kernel). This is in contrast to the UNIX approach whereby every process has its own unique address space. This has several benefits: it improves and simplifies data sharing, helps cache performance, and blurs the distinction between shared memory and distributed memory machines. The SASA is maintained between multiple processors using shared memory techniques. The SASA has become feasible with the appearance of large address space processors [17], enabling many processes to consume addresses from the same range without exhausting the supply.

This address space is managed as persistent objects (contiguous groups of pages). Not only does this remove the need for an explicit "file system" interface (with a different namespace and explicit system calls) but greatly simplifies the storage of complex structures, databases, etc.

3.2 Protection Issues

In Unix one process is protected from another by the use of separate address spaces. In a SASA all processes share the same address space, so separating protection from address translation, and hence a new scheme is needed to provide protection. This has also caused some researchers to propose alterations to the traditional memory and protection hardware with the addition of new hardware support [18]. The protection scheme must define two areas within which it works: the unit of protection, and the method used to specify and meet access requirements.

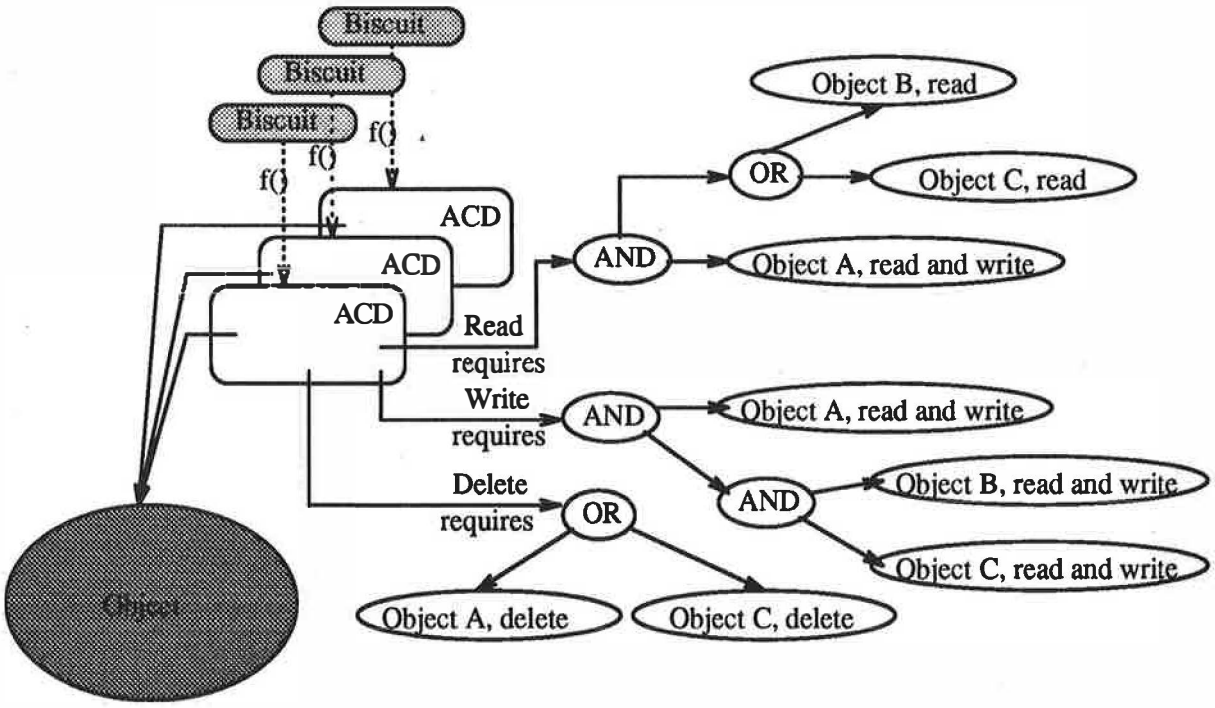


Figure 1: The structure of ACDs and biscuits

In *Angel*, protection is provided on objects which consist of one or more pages. Objects cannot overlap, nor may they be contained within other objects. A critical server in *Angel* is the object manager which is responsible for allocating addresses to objects and for validating access to objects. For every object, the object manager associated with it one or more *Access Control Descriptors (ACD)* which describe the other objects that must be accessible before this object may be accessed.

An example of this structure is shown in figure 1 in which one object has three ACDs associated with

it; one of them has part of its permissions tree show. In this example, to gain write access to the object, the process must already have read and write access to objects A, B and C.

When an object is created, or when a new ACD is associated with an object, the object manager gives out a *biscuit* from which it can reliably identify the valid corresponding ACD. Conceptually there is only one biscuit per ACD despite processes being free to duplicate this as frequently as they like. When a process wishes to access an object, it presents this *biscuit* to the object manager. The *biscuit* is then used to determine if the process possesses the necessary objects to resolve the requested object. Consequently, the system does *not* have the concept of user identifiers. However, it is trivial to implement such a system by creating an object whose sole purpose is to act as a "user id" for access checking.

3.3 Support for Parallel Programs

Angel supports first class threads and uses upcalls for inter-process and kernel-process signalling (see section 4.2). Their purpose is to allow process to be informed external events in which they have declared an interest, eg. the release of locks, the arrival of new work, a page fault or a pending time slice. By passing such information onto the process, the process is able to make its own decisions on what to run and to take remedial action (e.g. release a lock) when decisions are imposed upon it.

The DSM supported by *Angel* allows the construction of locks such as spin locks with reasonable efficiency. When combined with the upcall mechanism, it is simple for a thread to "sleep" and be "woken" at some later date. This provides asynchronous systems, not possible with shared memory alone.

The SASA that lies at the heart of *Angel* makes implementing load balancing trivial. As all processes and threads on all processors exist within a single address space that also contains all the necessary kernel information, moving a process or thread from one physical processor to another simply involves loading the processor context for the thread into the new processor. The DSM that implements the SASA will then move any necessary data as it is accessed. The design of *Angel* as it stands will not automatically load balance work for a process, but this can easily be provided through library routines.

3.4 The Angel Model

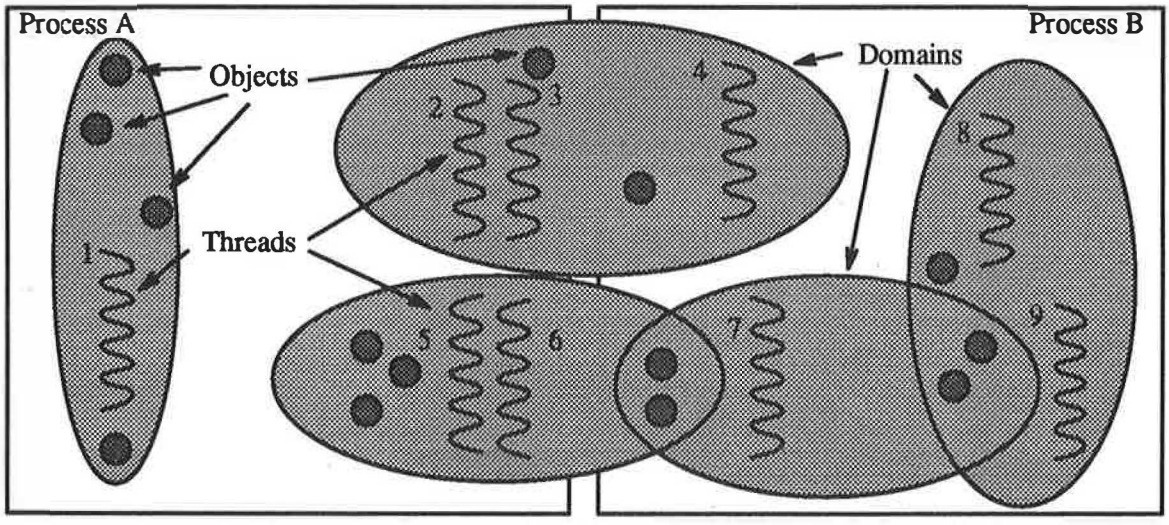


Figure 2: The Angel Process Model

Figure 2 shows how the above points are combined into the process model that *Angel* supports. Within any process there may be one or more threads. Threads may run in their own domain, as is the case with thread 1 in the diagram. This allows several threads in the same process to be protected from each other. Alternatively several threads may share a protection domain, potentially between different processes, as is the case with threads 2 to 4. Where threads do not wish to share a protection domain for security or trust reasons, they may have some mutually shared objects, as is the case with the remaining threads.

3.5 Fault tolerance

It is possible to build a scalable, efficient fault tolerance scheme in a SASA based operating system. This relies on the unification of resources to simplify the implementation, and the augmentation of the DSM system in order to capture the data interactions necessary to make distributed checkpoints. Unlike other schemes [19, 20] where excessive DSM activity can result in large number of checkpoints being made, we allow general data sharing without checkpoints, instead utilising the DSM state information to determine which data depends on which. This allows distributed checkpoints to be made which will only effect processes which are interacting, and also allows the DSM mechanism to be reused for checkpointing data to other machines' memories. Experiments indicate this costs only an additional 10% on an applications execution time. A full description can be found in [21].

4 The Angel implementation

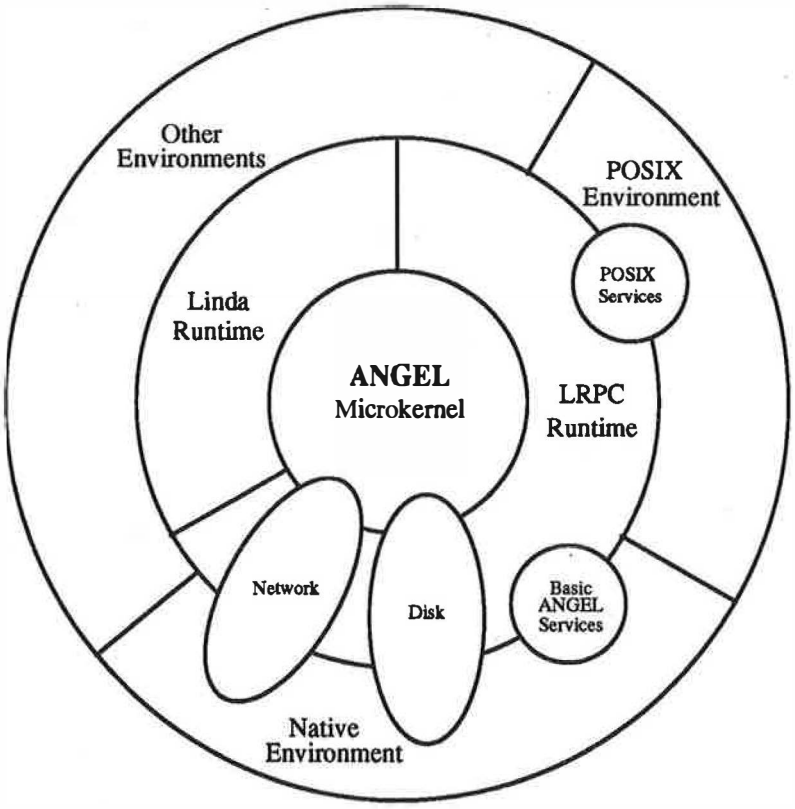


Figure 3: Angel structure

Figure 3 depicts the general structure of the *Angel* operating system. This structure has few differences

from more conventional, message passing microkernel designs. However, the use of a single address space and shared memory for communications has significantly simplified the microkernel. Currently, the implementation consists of 2,500 lines of C++ code and 1,000 lines of include files. This constitutes the virtual memory, the distributed shared memory and the device management systems but not the device drivers themselves.

At time of writing, we have completed initial work on the microkernel and client/server communication system. The microkernel provides two major services:

1. Persistent virtual memory, and
2. Virtual processor management.

The client/server communications are implemented using "lightweight" RPCs.

4.1 Persistent virtual memory

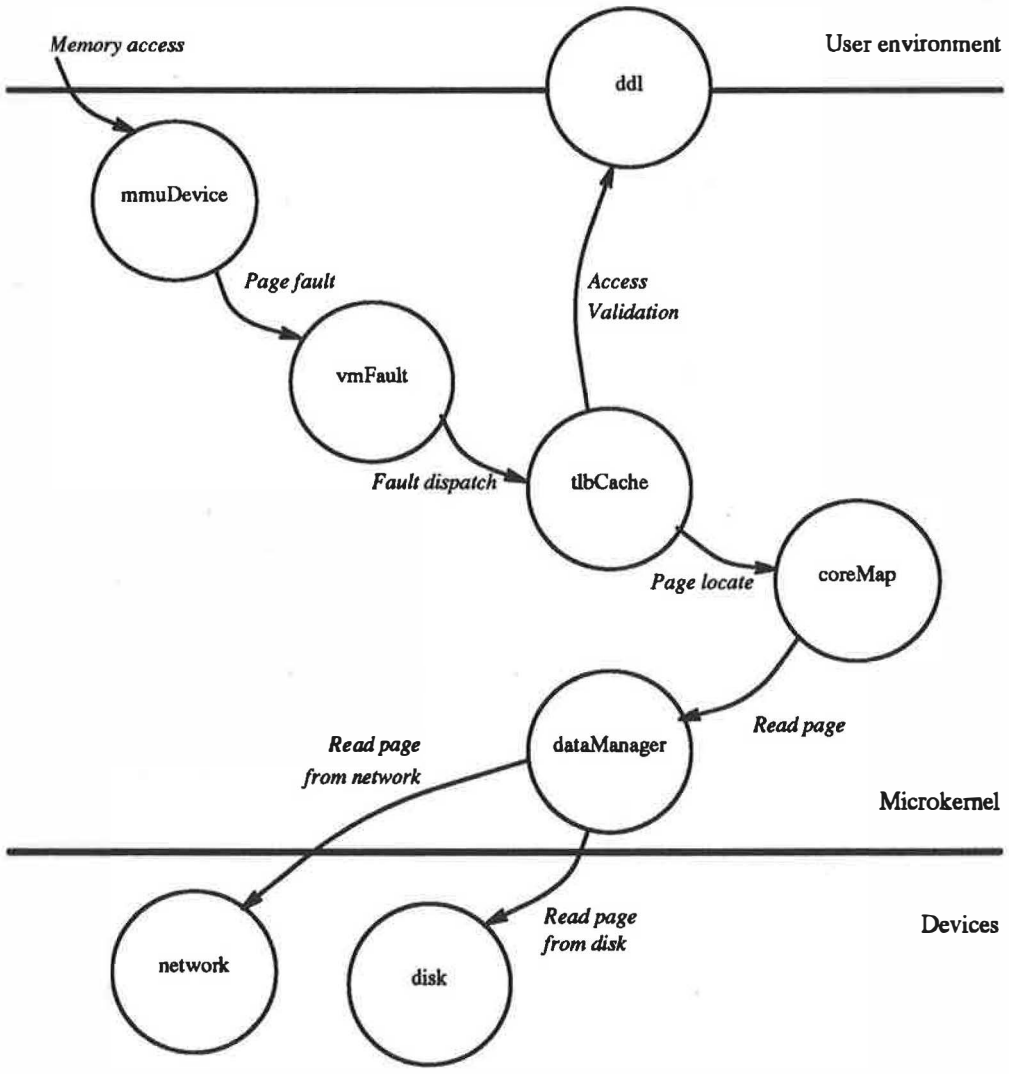


Figure 4: Object orientated VM system

The virtual memory (VM) system is the heart of *Angel* since it supports the persistent single address space. The single address space nature of the VM enables some simplifications of the structure to be made but the persistence introduces other complications.

Figure 4 demonstates the events in the VM system initiated by a page fault. Page faults are generated by the *mmuDevice*, a processor dependent object responsible for collecting all necessary information regarding the fault, and passed into the main, processor independent code, *vmFault*. This determines whether the fault is legitimate (user attempts to access supervisor data are caught here) and requests the relevant page from the *tlbCache*. The *tlbCache* first determines whether the access was to an object accessible by the virtual processor (using the *ddl* which describes this relationship). If it was not, a fault condition is returned. If it was, the accessed address is used to form a *pageID*, an unique identifier for data in time and space. These pageIDs are used to support data aliasing[2] necessary for the copy-on-write mechanism. The pageID is then used by the *coreMap* to locate the relevant data. The local coreMap memory is first searched for data corresponding to this ID. If found, the page is returned for installation by the mmuDevice. If not found, the coreMap allocates an empty core page and request the *dataManager* to find the data and install it. The dataManager does this by consulting both the *network* (which provides the DSM system) and the *disk*.

Several point in this VM system are worth special mention. First, the *ddl* is held in the user environment, so allowing it to be treated as any other object, sharable via the DSM and swappable onto disk. This prevents consumption of valuable kernel resources and allows the user to easily determine attributes of their envionment without the microkernel's assistance[3]. Second, the devices (network and disk) are accessed through an LRPC interface (see section 4.3). This allows them to be installed externally from the microkernel if desired although the LRPC mechanism will automatically optimise this interface when this is not the case. Currently, these devices are contained within the kernel but we are planning to make them loadable kernel-level device drivers in order to improve modularity and flexibility without compromising performance. Third, at various stages, the VM system may reach a point where it cannot continue immediately. This may be the result of a fatal error (eg. an access is made to an object not available to the user) or a temporary error (eg. the requested data must be fetched from disk). In these cases, the error is reported back to the virtual processor by use of an **upcall**. This enables the virtual processor to reschedule another thread.

4.2 Virtual processor management

The microkernel attempts to impose little process structure on the application or programmer. Unlike POSIX therefore, it does not implicitly provide such services as file descriptors, "death of child" signals or other heavyweight features. Consequently the process structure, termed a **virtual processor (VP)**, leaves much of the general management work to the application. This presents no additional problem since it can be encapsulated in libraries.

A virtual processor operates around two general data structures; its domain descriptor list (*ddl*) and its upcall list. The *ddl* holds information about all object the virtual processor has access to. As already mentioned, this object is used by the virtual memory system to determine the validity of memory accesses. However, it also holds information for processor management; such as which objects may be signalled using upcalls, and which object was initally executed.

The *upcall list* is the virtual processors' interrupt mechanism and is used by both kernel and other VPs for preempting each other when important events occurs. These events include:

- Alarms,
- Invalid memory accesses,
- Temporarily invalid memory accesses, and
- Lock releases.

2. This is where two or more virtual addresses reference the same physical data.
3. Natually, the user is prevented from direcly modifying the ddl.

The first three of these events are microkernel generated; the forth is generated by user level code associated with the release of mutual exclusion locks or conditional variables.

Upcalls are a fixed sized structure, convey little information, and will not be delivered if the recipient has insufficient resources to receive them. Each one identifies its sender, its type and two further type specific pieces of information (eg. Invalid memory accesses report the failed address and reason for the failure; lock releases report the address of the locking structure.). The VP can precisely control the effect each upcall has when it delivered, determining whether a handler is invoked immediately, whether the upcall is queued for later attention, or whether the upcall is ignored completely. By default, all upcalls are ignored unless the VP specifies otherwise. This generally means that upcalls are simply discarded without effect although "invalid memory accesses" will terminate the VP.

4.2.1 Threaded virtual processes

Angel does not explicitly support threaded processes, leaving this to user level code. However, through the use of kernel and user level upcalls, it still provides facility for a "first class citizen" thread model. For example, in the kernel, whenever a situation occurs where it should block, the VP is upcalled to allow another threads to be scheduled. Similarly, user level locks can use this facility in parallel programs or client/server relationships (we use this heavily in the LRPC mechanism). At the user level, a POSIX thread model [22] is provided. The operation of POSIX threads is well documented, but it is worth nothing how this model interfaces to *Angel*'s upcall system in order to provide "first class citizens".

All locks are implemented in shared objects. For mutual exclusion locks, if a lock is not obtained, the failed thread inserts itself into the lock's pending queue. The thread scheduler is then called to dispatch another, the VP blocking if there are no others ready to run. When the lock is released, the releasing thread examines the head of the pending queue and releases the top thread. If this thread is within the same protection domain, the operation can be accomplished locally. If not, a *lock release upcall* is dispatched to the appropriate VP. On receiving this, the thread is released locally. The mechanism used for conditional variables is similar to this except that the thread release is delayed until the associated lock is released. By placing locks in shared memory, the operations of obtaining and releasing locks is greatly simplified and the need to consider whether a thread is local or remote is hidden.

4.3 Client/Server Communications

Like many commercial and research operating systems, *Angel* uses the notion of clients and servers in order to improve the functional modularity of the system. However, unlike many of its predecessors, message passing is not used to implement RPC communication, instead this is done through shared memory regions. This approach enables a more "lighweight" RPC mechanism to be implemented (based on work by Bershad et al [5]).

Angel's LRPC mechanism operates by the sharing of C++ objects in sections of shared memory. These objects are passed between client and server by manipulation of shared lists and the release of the associated locks. However, optimisations in this mechanism are possible if both client and server operate in the same protection domain. In such cases a direct subroutine call can be made from client to server so avoiding the need for locking altogether. This optimisation can be determined when the LRPC channel is established rather than at compile time so providing greater flexibility.

4.3.1 LRPC example

Figure 5 illustates a simple client/server interaction using a shared memory object for communication. This object constitues a private channel between parties, available in their protection domains only (although one-to-many channels are no more difficult to arrange).

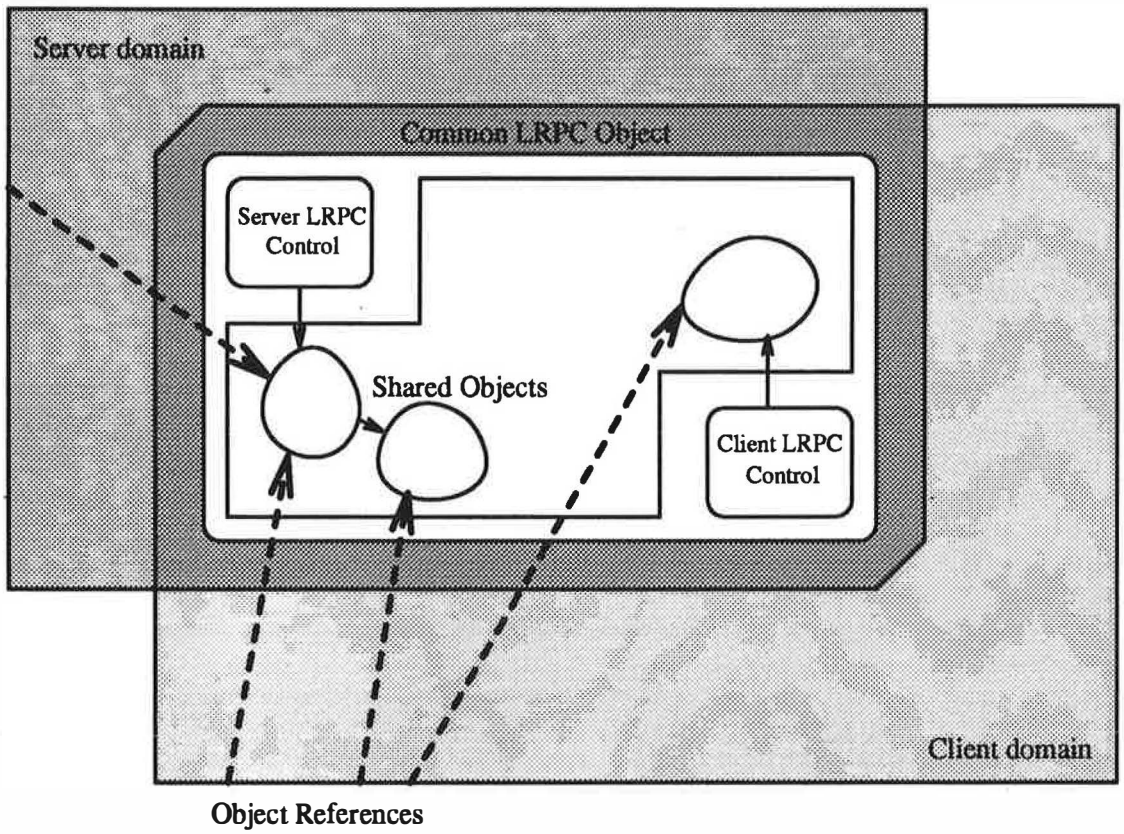


Figure 5: Lightweight RPC object shared between a client and server

In conventional RPC, a client makes a request of the server by packaging data to be transfered and then informing the server of its intentions. The server then unpackages the request, performs the work, and replies to the client using a similar RPC mechanism. LRPC in *Angel* benefits over such a system in two ways; first, the use of shared memory reduces the need to package data, in some cases removing it altogether; and second, implict encapsulation of the client/server relationship in C++ classes simplifies and hides access to the interface.

For example, the server in figure 5 maintains the private database holding users' information. A client wishing to search this database (such as `/bin/ls -l`) must make requests via an LRPC channel. However, rather than constructing and copying requests to the server, a `passwdEntry` object can be allocated which is already shared with the server using the C++ placement operators (eg. overloading of `operator new()`). This object can then be used as normal within the client, the interaction with the server happening transparently and without extra copying by either party.

4.4 Current status

The majority of development work has been done by operating the microkernel as an emulation under SunOS UNIX. However, in order to validate the system and determine whether our efforts to keep the dependent and independent code seperate have been successful, we recently ported the kernel to a Tadpole M88K system. This work took a week to complete despite the need to write a new two-level MMU system and although some restructuring has resulted, no major problems were encountered.

However, neither of these systems are appropriate to *Angel*'s needs due to the restricted address space. Currently we are investigating a port to either an SGI Indigo or DEC Alpha PC either of which is more

appropriate.

5 Lessons and Further Work

The most "politically difficult" decision to make regarding *Angel* was to forego UNIX compatibility. It is acknowledged that if an SASA style operating system is to accepted, then it must provide support for UNIX and its existing software base. As a first step we have investigated modifying compilers to generate code that gave the appearance of UNIX memory semantics. This resulted in a performance penalty of only a few percent [23]. We are now investigating a full UNIX service under *Angel*. It appears that a reasonable degree of compatibility can be provided at low cost, without altering the SASA to provide a region of memory addresses with UNIX characteristics.

The fault tolerance mechanism described above has been designed, implemented and analysed on a simulator, rather than in the current *Angel* implementation. One, relatively simple, task is therefore to implement this scheme in the current microkernel. Once this has been done we hope to study the performance of the system and see if it can be further improved.

The main area of future work lies in dealing with the projected large I/O requirements that a parallel computer will generate. Many current parallel computers are badly I/O limited, and overcoming this bottleneck is extremely important in opening up new markets for parallel machines. There are several schemes we are currently investigated to perform this, the most hopeful is to make use of the algorithms from the fault tolerance scheme which generates a distributed log stream of data for storage on disk.

6 Conclusions

This research was conceived as an exercise in learning from *Meshix* (and other message passing micro-kernels); the result is the Angel operating system, which is still a micro-kernel, but is based around a SASA supported by DSM, and not around message passing. The current implementation is small, and has been easy to write, which leads us to believe that we have constructed a good design, and that a SASA is the way to build systems. There are other benefits from this approach which are important to scalability, for example in the areas of fault tolerance, data sharing and load balancing. Although we have not developed the system with UNIX support it mind, it appears that we can provide a simple version of this at very low overheads. All these points lead us to believe that SASAs are an important way of constructing operating systems, especially for scalable, parallel machines.

7 Authors Information

Dr Kevin Murray's thesis work at the University of York concerned the development of Wisdom, an operating system designed to support a high-level programming environment on a DMMP conforming to a subset of the ANSA transparency model. In addition, he contributed to the development of Wisdom's filesystem. He then worked at Imperial College, in collaboration with the Systems Architecture Research Centre, on the Angel operating system concentrating on its scheduling and inter-process communications aspects, before being appointed lecturer at City University, where he has remained heavily involved in the Angel work.

Tim Wilkinson has worked extensively on the Topsy project including work on the Meshix OS and Meshnet communications chips. His PhD work, now nearly completed, centres around the design of a reliable 64-bit distributed operating system using data dependent checkpoints. He is currently employed on the Angel operating system project.

Prof. Peter Osmon is head of the Systems Architecture Research Centre at City University. He was Principal Investigator on the Alvey-funded Cobweb project. He conceived and directed the Topsy Unix multicomputer project. He has a current IED grant with Phoenix VLSI and Texas Instruments concerned with the design of an interface device to support shared-memory over a serial interconnect (ICTVS, reference number GR/F99618), and he is Principal Investigator of the SERC funded project developing the Angel kernel (GR/G28277).

Dr. Tom Stiemerling has worked on implementing DVSM on Topsy, and the specification and implementation of the Angel kernel, and is supported by SERC research grant GR/G 28277. His doctoral work carried out at Edinburgh University involved the performance analysis by simulation of a shared memory multiprocessor architecture.

Dr. Paul Kelly is a lecturer in the Department of Computing at Imperial College. He was a researcher on the Alvey-funded Cobweb project. His doctoral work led in part to IED projects on functional programming of transputer networks, and exploitation of more general parallel hardware using functional languages and program transformation. More recently he has collaborated in the development of Paragon, an object-oriented graph-rewriting language, and is also an investigator on the related SERC-funded project developing the Angel kernel at Imperial (GR/G23562).

Bibliography

[1] Open Software Foundation, "The OSF/1 operating system," in *Spring 1991 EurOpen Conference*, pp. 33–41, 1991.

[2] M. Rozier, "Overview of the CHORUS Distributed Operating Systems," Tech. Rep. CS-TR-90-25, Chorus Systemes, 1990.

[3] P. Winterbottom and P. Osmon, "Topsy: An Extensible Unix Multicomputer," in *UK IT90 Conference, Southampton University*, 1990.

[4] A. Bricker, "A new look at micro-kernel-based UNIX operating systems: Lessons in performance and compatability," in *EurOpen Spring'91 Conference, Tromsoe, Norway*, May 1991.

[5] B. Bershad, T. Anderson, E. Lazowska, and H. Levy, "Lightweight remote procedure call," *ACM Operating Systems Review*, vol. 23, pp. 102–113, December 1989.

[6] P. Osmon, T. Stiemerling, A. Whitcroft, Wilkinson.T., and N. Williams, "Evaluating Meshix – a Unix compatible micro-kernel Operating System," in *OpenForum'92*, November 1992.

[7] A. Whitcroft and P. Osmon, "The CBIC: Architectural Support for Message Passing or Shared Memory?," in *U.K. Performance Engineering Workshop*, September 1992.

[8] K. Li, *Shared Virtual Memory on Loosely Coupled Multiprocessors.* PhD thesis, Yale University, Department of Computer Science, 1986.

[9] U. Ramachandran, G. Shah, S. Ravikumar, and J. Muthukumarasamy, "Scalability study of the KSR-1," Tech. Rep. GIT-CC93/03, College of Computing, Georgia Institute of Computing, Atlanta, Georgia, 1993.

[10] E. Hagersten, A. Landin, and S. Haridi, "DDM – A Cache-only Memory Architecture," Tech. Rep. Research Report R91:19, SICS, Sweden, November 1991.

[11] M. Hill, J. Larus, S. Reinhardt, and D. Wood, "Cooperative shared memory: software and hardware for scalable multiprocessors," in *ASPLOS V*, pp. 262–273, September 1992.

[12] J. C. Mogul and A. Borg, "The Effect of Context Switches on Cache Performance," in *ASPLOS, International Conf. on Architectural Support for Programming Languages and Operating Systems*, (Santa Clara, CA (USA)), pp. 75–85, April 1991.

[13] B. Marsh, M. Scott, T. LeBlanc, and E. Markatos, "First-Class User-Level Threads," Tech. Rep., Computer Science Department, University of Rochester, NY, 1991.

[14] E. Organick, *The Multics system: an examination of its structure.* M.I.T. Press, 1972.

[15] M. Scott, T. LeBlanc, B. Marsh, T. Becker, C. Dubnicki, E. Markatos, and N. Smithline, "Implementation Issues for the Psyche Operating System," Tech. Rep., University or Rochester, Department of Computer Science, 1988.

[16] J. Chase, H. Levy, M. Baker-Harvey, and E. Lazowska, "How to Use a 64-Bit Virtual Address Space," Tech. Rep. 92-03-02, Department of Computer Science and Engineering, University of Washington, March 1992.

[17] Dobberpuhl *et al.*, "A 200Mhz 64-bit Dual Issue CMOS Microprocessor," in *International Solid-State Circuits Conference*, February 1992.

[18] E. Koldinger, J. Chase, and S. Eggers, "Architectural support for single address space operating systems," in *ASPLOS V*, pp. 175–186, September 1992.

[19] K.-L. Wu and W. Fuchs, "Recoverable distributed shared virtual memory," *IEEE Transactions on Computing*, vol. 39, pp. 460–469, April 1990.

[20] B. Fleisch, "Reliable distributed shared memory," in *IEEE Workshop on Experimental Distributed Systems*, pp. 102–105, 1990.

[21] T. Wilkinson, "Implementing Fault Tolerance in a 64-bit Distributed Operating System," Tech. Rep., City University, 1993.

[22] POSIX 1003.4a, "Threads Extension." IEEE Draft.

[23] T. Wilkinson *et al.*, "Compiling for a 64-Bit Single Address Space Architecture," Tech. Rep. TCU/SARC/1993/1, SARC, City University Computer Science Department, March 1993.

Experimentation with a Reconfigurable Micro-Kernel

Bodhisattwa Mukherjee *Karsten Schwan*

College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332
e-mail: {bodhi, schwan}@cc.gatech.edu

Abstract

Since the implementation of operating system functions can significantly affect the performance of parallel programs, it is important to customize operating system functionality for specific application programs. In this paper, we propose an architecture for a reconfigurable micro-kernel. This kernel can be configured at compile-time and at execution-time to suit varying application requirements. Such a micro-kernel can be used for the development of high performance operating systems and applications for parallel and distributed systems. We have implemented the reconfigurable micro-kernel on multiple parallel machines, including a 32-node GP1000 BBN Butterfly, SGI multiprocessors, and a 32-node Kendall Square Supercomputer.

1 Introduction

Earlier research in operating system for parallel and distributed systems has demonstrated the need for reconfiguration and customization of operating system mechanisms to suit each class of application programs. For example, the set of functions provided by HYDRA[25] was designed to enable the user of C.mmp to create his own operating environment without being confined to predetermined mechanisms and policies. Similarly, for real-time applications executing on shared memory multiprocessors, the object-based operating system kernels described in [9] offer several representations of objects and object invocations to support the different degrees of coupling, task granularities, and invocation semantics existing in real-time applications[9]. Such experiences in the real-time domain are mirrored by work in multiprocessor scheduling[3] that demonstrates the importance of using application-dependent information or algorithms while making scheduling decisions. In addition, tradeoffs in program performance due to the use of alternative synchronization constructs have been demonstrated for various parallel architectures, including the experimental C.mmp[25] and Cm*[8] multiprocessors, interconnection-network-based machines

like the Ultracomputer[22], and bus-based machines like the Sequent Symmetry[1]. For example, for bus-based UMA machines, Anderson showed that spin locks can put a significant load on the shared bus, so that efficient use of the parallel machine requires a back-off strategy for spin locks similar to the one used by low-level Ethernet devices[1]. Such a back-off strategy is a typical example of dynamic reconfiguration since it dynamically alters the back-off delay. Lastly, recent research addressing efficient memory models for shared memory and for distributed memory[23] machines has made it clear that the support of multiple semantics of memory consistency can result in improvements in parallel program efficiency.

Developing micro-kernel based operating systems for parallel and distributed systems has been a recent thrust in operating system research, and it has led to systems structured as collections of user-level server processes or threads layered on a minimal kernel. Our research is complementary to such efforts, wherein we investigate the reconfigurability of different abstractions implemented as part of the micro-kernel itself. Specifically we investigate:

- How can a micro-kernel support multiple configurations (*configurability*)?
- How can a micro-kernel be structured so that it can be reconfigured to suit the changing runtime requirements of application programs (*reconfigurability*)?
- Is such dynamic reconfiguration advantageous? Namely, do the runtime costs incurred by dynamic reconfiguration justify the possible gains due to such reconfiguration (*adaptability*)?
- What functionality is required to build an adaptive micro-kernel?

The next section first discusses a few sample examples of reconfiguration in operating systems, then presents the structure of the operating system kernel being built by our group. Section 3 describes the basic architecture of a reconfigurable micro-kernel. A few measurements of the basic micro-kernel functionalities are presented in Section 4. Section 5 compares our work with related research and finally, section 6 concludes the paper and presents some future directions.

2 Reconfiguration in Operating Systems

The feasibility of dynamic reconfiguration has already been demonstrated in the real-time domain[9], and for parallel applications regarding their runtime synchronization on large-scale multiprocessors. Specifically, past research demonstrates that a single parallel program exhibits dynamic changes in its locking pattern during execution[17] (*i.e.*, , it exhibits changes in the distributions of locking requests and of the number of threads waiting for a lock). Therefore, it is possible to utilize information about such patterns to dynamically change the behavior and consequently, the runtime performance of the synchronization

primitives implemented as part of the operating system kernel. Similarly, earlier research demonstrates that dynamic reconfiguration is often useful in process/thread management and scheduling. For example, dynamic process migration and load balancing are often used in existing operating systems for improved performance. Such dynamic load balancing techniques (a static load balancing technique monitors the load of a processor only at the process creation time, and does not migrate the process after initial placement) monitor the load of individual processors and migrate threads or processes from one processor to another at run-time.

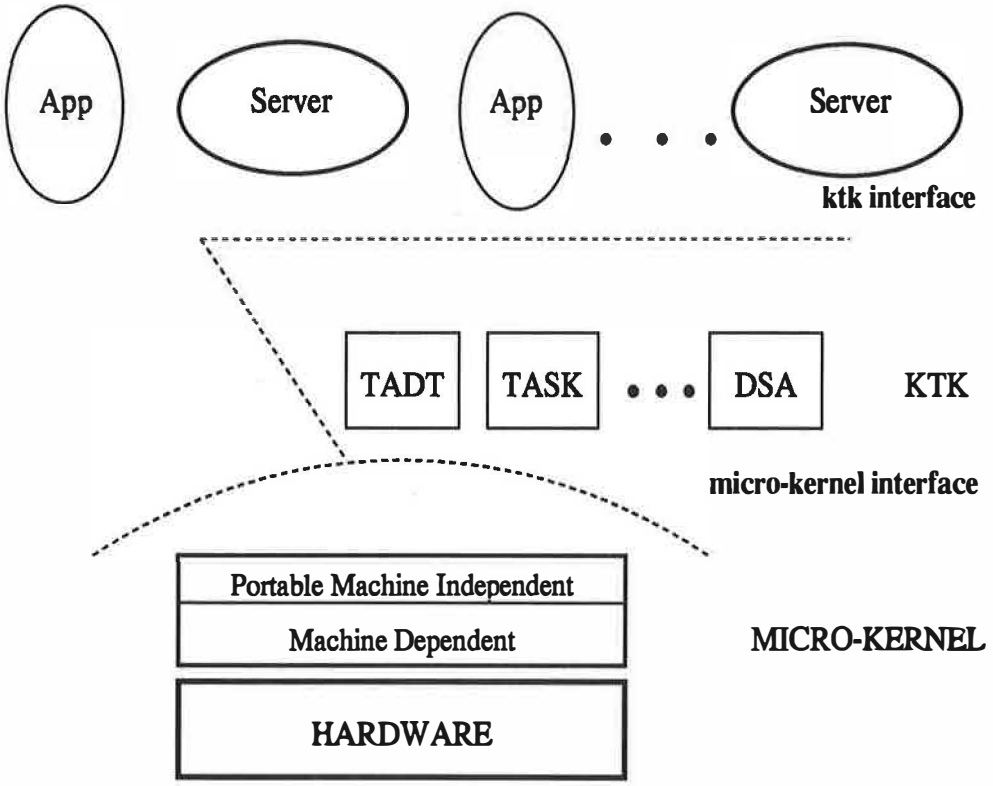


Figure 1: System Structure

Recent studies show that improvements in processor speed and therefore, application code performance, are not matched by similar improvements in performance of basic operating system functions (system call, trap, page table entry change, context switch *etc.*) [2]. We argue that application performance can be improved significantly by matching operating system kernel functionalities with specific application characteristics and specific hardware. Specifically, we posit that:

- The run-time behavior of applications differ across multiple applications and across multiple phases of a single application. (This is demonstrated by experimentation in [11, 17])

- Operating system kernel configurations can provide high-performance applications with the policies and mechanisms best suited to their characteristics and to their target hardware.
- Dynamic kernel reconfiguration can improve performance by satisfying each application's changing behavior.
- Efficient application state monitoring can detect changes in application requirements which are not known prior to program execution time.

As stated above, the object of our research is to improve operating system performance by using application information to select and/or customize the operating system facilities used by the application program. We are constructing a dynamically configurable kernel – the Kernel Tool Kit (KTK)[9] that offers the flexibility to customize and/or reconfigure its mechanisms and policies to suit an application's runtime needs and the underlying multiprocessor hardware. The Kernel Toolkit (KTK) provides support for the construction of operating system kernels suited to specific classes of application programs. KTK offers a small set of abstractions (built-in object classes) and policies that have been extended and customized for several parallel application programs, including real-time programs like those required for autonomous robotics[20, 9], highly dynamic parallel applications like parallel branch-and-bound codes[21], and numerical simulations. KTK is also the basis for our current research addressing configurable communication protocols for parallel and real-time systems[13].

As shown in Figure 1, the whole system is based on a micro-kernel which consists of two well-defined portions – a machine dependent portion which encapsulates the underlying hardware, and a portable machine independent portion built on top of the interface provided by the machine dependent portion. The micro-kernel provides a basic set of functionalities primarily at the threads level, whereas, among other things, KTK (the object layer)[9] offers the notions of classes,objects, and invocations on top of the micro-kernel. It supports various types of objects. For example an object of class TADT (threaded abstract data type) is active and is used for the representation of parallelism in applications. Similarly, a TASK object is like an Ada task in that it consists of a single active thread of control. KTK also provides support for fragmented objects using a distributed shared abstraction (DSA) module[7]. This module:

- permits programmers to define and create encapsulated, fragmented objects, and
- offers low-level mechanisms for implementing efficient, abstraction-specific communications among object fragments.

The KTK interface and/or the micro-kernel interface can be used to build applications and operating systems (by means of servers) suited for specific application class and hardware.

Past research in dynamic reconfiguration of parallel computation for improved performance has been primarily concentrated either at the application level or at the object level in the operating system. For example, MACH[4] and CHORUS[19] support operating system reconfiguration at the server level. CHOICES[6] operating system supports reconfiguration at the object level. PRESTO[3] supports reconfiguration at the application level. Similarly, KTK[9] supports object reconfiguration (using attributes, and a special kind of object called policies) and various flavors of invocation at the object level. We argue that it is more appropriate to incorporate reconfiguration at the micro-kernel level:

- as the operating system and the applications are layered on top of the micro-kernel, both layers will benefit from any performance improvement caused due to reconfiguration at the micro-kernel level.
- the micro-kernel encapsulates the hardware and so is the most suitable candidate to capture hardware reconfiguration (*e.g.*, network reconfiguration, device configurations *etc.*) and changes in hardware.

The next section describes the architecture of such a reconfigurable micro-kernel.

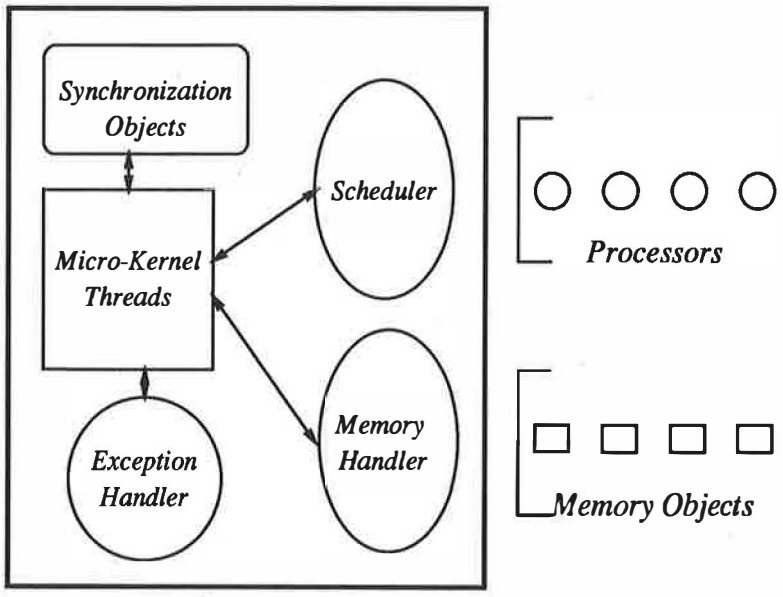


Figure 2: Micro-Kernel Components

3 The Basic Architecture

The basic components (Figure 2) of the micro-kernel being developed by our group are:

1. Processor and thread management

2. Low-level memory management

3. Thread synchronization/communication management

4. Minimal exception handling

5. Customized monitoring support

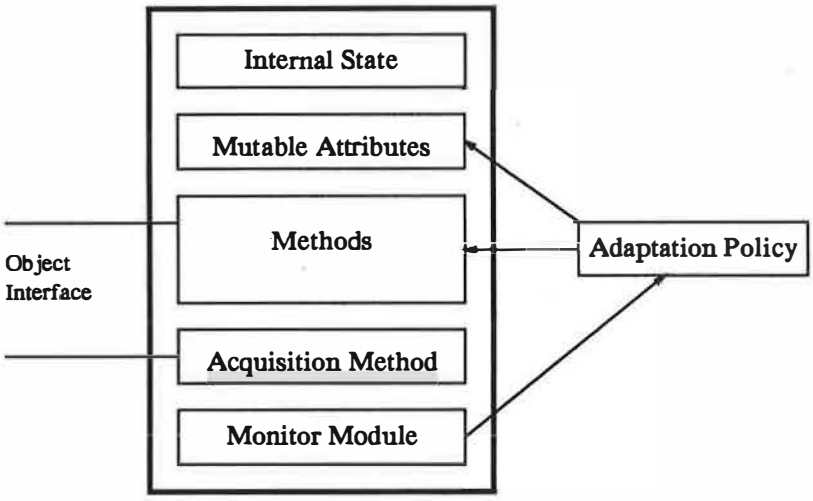


Figure 3: Structure of an Adaptive Kernel Component

The micro-kernel provides specific support for configurability, reconfigurability, and adaptability of its different components:

- It defines a set of abstraction-specific *attributes*. We are concerned with attributes that characterize a component's internal implementation such as its data structures and some of its methods. Such attributes are altered dynamically by applications explicitly or implicitly by a "user-provided adaptation policy". We assume that changes in attributes may be performed both synchronously or asynchronously with method invocations. This requires the introduction of two additional time-dependent properties of attributes: (1) attribute mutability and (2) attribute ownership. An attribute is *mutable* whenever its current value may be changed. For example, a lock object's attribute specifying its waiting policy (not its scheduling policy) is permanently mutable because it may be changed at any time. However, its scheduling policy is likely to be immutable whenever threads are waiting on the lock, due to the inordinate potential expenses involved with the reorganization of internal lock data structures like thread waiting queues[17].

- It provides support for efficient customized state monitoring. Such information (about kernel components) may be used at the application level in the form of an "adaptation

policy" to build adaptive operating system[17] components. For example, a simple adaptive lock samples the lock contention (in terms of number of waiting threads) periodically at a user-defined rate, and adjusts its implementation-specific attributes (such as `spin-time`, `sleep-time`, `release` *etc.*) according to an adaptation policy.

- It permits an application to define adaptation policies that guide the reconfiguration of the kernel components, it uses.

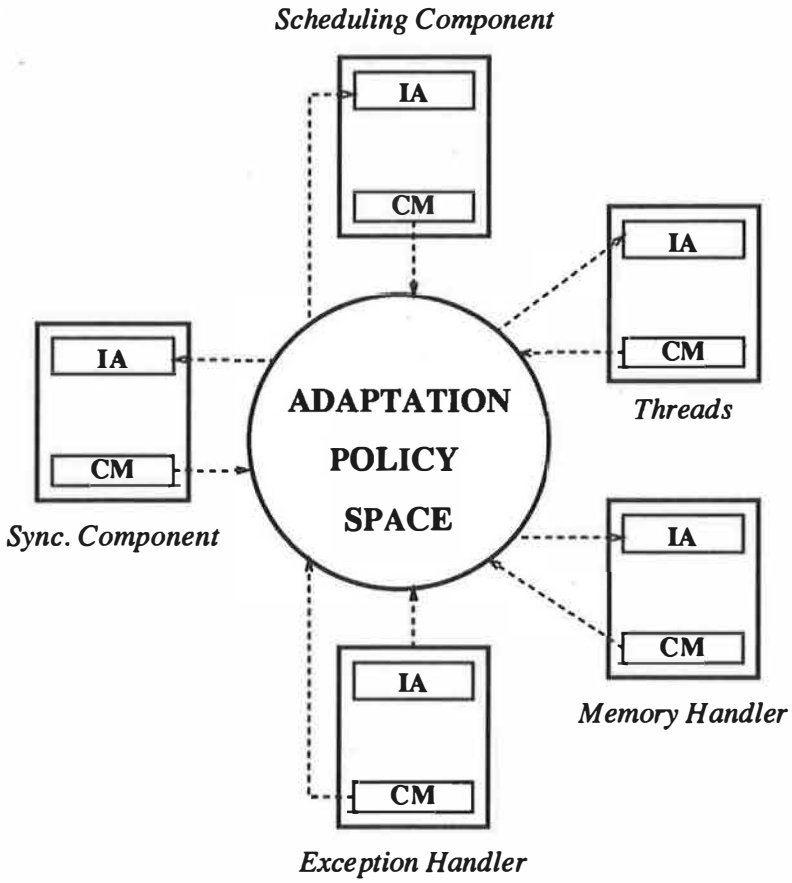


Figure 4: Micro-Kernel Structure

Figure 3 shows the structure of a "closely coupled" reconfigurable kernel component[17]. Such components provide mechanisms to alter the implementation of one or more of their methods dynamically (at execution time). A reconfigurable component consists of its state, a set of mutable attributes, a set of acquisition methods (used by an external agent to explicitly acquire one or more mutable attributes for reconfiguration), a customized monitor module (to monitor a set of predefined state variables, and attributes), and an adaptation policy (to guide the reconfiguration operation) [17].

The reconfigurable micro-kernel consists of a collection of such adaptive/reconfigurable components. Figure 4 illustrates the conceptual structure of the micro-kernel. Each compo-

nent contains a set of implementation dependent attributes (IA) and a customized monitor module (CM). The reconfiguration of each individual component is guided by adaptation policies which constitute the "adaptation policy space". A few sample adaptation policies are described elsewhere [17].

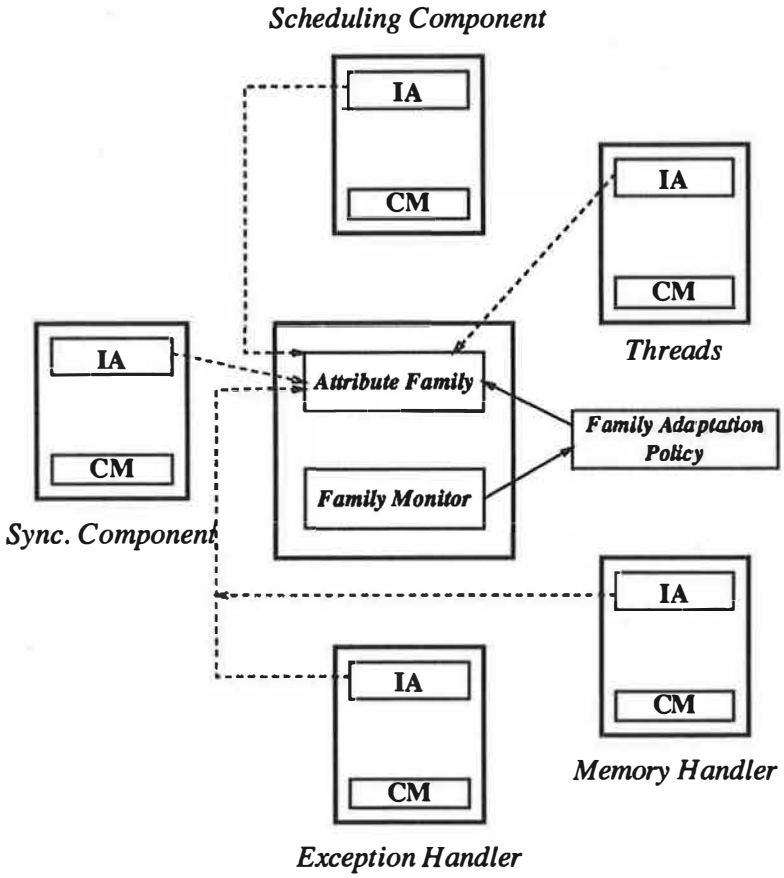


Figure 5: Attribute Family

As demonstrated by earlier research, individual abstractions cannot often be configured independently. For example, Ousterhout[18] demonstrated that co-scheduling considerably decreases inter-thread communication/synchronization overhead. Similarly, lock configurations, cache affinity *etc.* affect scheduling overhead[26] and scheduling policy, which in turn, affect thread exception handling overhead. In NUMA multiprocessors, memory management policy has a significant impact on IPC overhead and scheduling policy. Hence, the reconfigurable micro-kernel provides support to jointly reconfigure a group of kernel components using the notion of *attribute families*. We define an *attribute family* to be a group, collection, or set of attributes belonging to more than one kernel component. The run-time support for attribute families includes simultaneous reconfiguration of more than one of its components, customized monitoring of attribute families designed to sense the

inter-abstraction interactions, support for user defined adaptation policy for simultaneous adaptation of multiple abstractions.

The remainder of this section briefly describes the individual components of the micro-kernel:

Micro-kernel threads: Like most other operating systems, the micro-kernel supports the thread abstraction for managing execution. It supports multiple threads of control within one address space by providing minimal support for thread management such as thread creation, termination *etc.* Each individual thread has its own program counter, its own stack and a predefined set of hardware registers (currently, all the hardware registers are used by each thread due to lack of support for register partitioning). At the micro-kernel level, threads communicate only via shared memory as opposed to the next layer – the object layer – which provides support for multiple mechanisms for inter-thread communication such as object invocations, message passing (DSA), and RPC. Currently, the threads component has only one attribute – the stack-size – which is configured statically.

Synchronization Component: The micro-kernel threads synchronize using reconfigurable locks[16, 17]. Such locks provide mechanisms for static and dynamic lock reconfiguration and permits applications to define their own adaptation policies. Reconfigurable locks contain two kinds of attributes – (1) *wait* attributes (`spin-time`, `sleep-time`, `delay-time`, `timeout` *etc.*) which implement a spectrum of lock waiting policies and (2) *scheduling* attributes (`registration` attribute logging all threads desiring lock access, `acquisition` attribute determining the waiting mechanism and policy to be applied to each registered thread and `release` attribute that grants new threads access to the lock upon release) which determine the way lock requests are served. The *scheduling* attributes of a reconfigurable lock determine the delay in lock acquisition experienced by threads, whereas, its *wait* attributes specify the manner in which a thread is delayed while attempting to acquire the lock[16, 17].

Scheduling Component: The scheduling component of the micro-kernel schedules application threads to allocated processors. The micro-kernel assumes the existence of pre-allocated cluster of processors for applications[24]. Earlier research demonstrates that parallel applications exhibit a wide range of scheduling requirements. For example, interactive applications require preemptive scheduling whereas search applications (*e.g.* branch-and-bound algorithms) perform better using non-preemptive scheduling with a specific application-dependent queue ordering scheme. The scheduling policies and scheduling mechanisms matching application requirements can differ widely, and can be expressed with a few dynamic implementation dependent attributes such as granularity, storage/retrieval policy, locality, preemption and hints.

The *granularity* attribute of a scheduler defines the scheduling unit. Group scheduling decreases interprocess communication cost, thus improving the performance of applications that require frequent thread communication[18]. However, group scheduling techniques exhibit higher scheduling overhead. Therefore, information regarding the pattern/frequency

of communication among the application's threads can be used to determine the appropriate granularity of the thread scheduler.

The *storage/retrieval policy* attributes of a thread scheduler specify the scheme used to store and retrieve threads to and from the thread repository (*e.g.*, queue ordering scheme for run queues and free lists). Different applications require different queue ordering schemes for enhanced performance. For example, optimization codes using search methods are easily mapped to algorithm-specific queue ordering schemes (*e.g.*, FIFO, LIFO *etc.*) whereas applications implementing prioritized services require priority queues. It should be apparent that the use of queue structures supporting specific storage/retrieval policies is superior in performance to the use of queue structures able to implement multiple policies.

The *locality* attribute of a scheduler determines the structure of the thread repository (e.g., centralized, distributed, etc.) and the load balancing strategies used in distributed repositories. Especially on the KSR machine, global queue organizations are not suitable for scalable performance in thread scheduling and thread management, so that queues must be internally fragmented across different processors[7]. Additional scheduler attributes must capture whether threads may be migrated across processors, which affects load balancing and cache performance. Alternative implementations of load balancing techniques have been demonstrated to be useful in multiprocessors to increase processor performance by decreasing processor idle time. However, certain applications exhibit performance degradation because load balancing strategies ignore cache-affinity of application threads [12]. Therefore, for enhanced performance, application characteristics will determine the locality attribute of the thread scheduler.

Scheduling policies may be classified as enforcing (1) involuntary preemption, which means that a scheduler can preempt a thread at any time, (2) voluntary preemption, which means that threads must release processors voluntarily, and (3) non-preemptive, which implies that threads are never preempted. Such differences result in major changes in the implementation of thread context switching (saving or not saving signal masks, for instance); they occur for real-time vs. non-real-time schedulers. *Preemption* introduces extra overhead in scheduling cost. However, interactive applications, master/slave programs *etc.* often exhibit improved performance with preemption.

Furthermore, in [5], the author demonstrates performance improvements of some applications (especially, the applications consisting of threads which frequently synchronize and/or communicate) using scheduling techniques that employ application-provided *hints*[1] when making scheduling decisions.

Even though the micro-kernel defines a collection of default schedulers such as non-preemptive FIFO scheduler, preemptive round-robin scheduler, priority scheduler *etc*, applications are encouraged to write their own customized schedulers (static configuration) for improved performance using the following framework:

[1] such as Discouragement hints and Hand-off hints

```
CLASS Scheduler is

        STATE internal_state is
                LOCK sched_lock;
                int node;
                int active_threads;
                QUEUE_T free_list;
                QUEUE_T *lqueue;
                THREAD_T scheduler_thread;
        END

        ATTRIBUTESET static_scheduler_attributes is
                int no_of_readyqs;
                int no_of_threads_per_proc
                HINT_T hint;
                OPERATION global_init (..);
                OPERATION vproc_init (..);
                OPERATION thread_init (..);
                OPERATION get_thread (..);
                OPERATION put_thread (..);
                OPERATION empty_readyq (..);
                OPERATION proc_idle (..);
                OPERATION schedule (..);

        BEGIN
        END
```

The above class defines an application-specific scheduler. The internal state of the scheduler provides data structures to define/implement multiple scheduling policies. The `attributeset` of a scheduler consists of various levels of initialization operations (`global_init` initializes at the processor level, `vproc_init` initializes at the virtual processor level, and `thread_init` initializes at the thread level), different policy operations (`get_thread` to get the next thread to run, `put_thread` to put a thread back to a bank of ready queues, `empty_readyq` to determine if a ready queue is empty, `proc_idle` to specify work for an idle processor), and a set of operations to handle various exceptions/interrupts (primarily used to implement preemptive scheduling based on inter-processor interrupts, timer interrupts etc.).

An application defines the scheduler(s) at pre-execution time and installs them to the specific processors using a call to:

```
install_scheduler(processor number, scheduler)
```

The performance of many applications can be improved if different, concurrently executable parts of their computations can be scheduled with differing thread schedulers –

termed *heterogeneous scheduling*. To demonstrate performance gains, we have studied distributed implementations of the "work queue" abstraction (implemented using the DSA module of the object layer) in a parallel branch and bound application. In this program, performance improves when using a non-preemptive FIFO scheduler for application threads and simultaneously using a preemptive priority scheduler for asynchronously executable threads implementing the communications among the different fragments of the work queue abstraction. Similarly, performance improvements are possible when a FIFO scheduler is used for application threads while using a priority scheduler for the transmission of monitoring data collected about the application during its execution.

Other Components: The exception handling component of the micro-kernel permits applications to define their own customized exception handlers. The micro-kernel also provides a set of default handlers for various exception conditions. The memory handler module provides support for low-level dynamic storage allocation and is configurable in that it permits applications to vary the minimum chunk of memory allocation for performance improvement.

4 Performance

A prototype reconfigurable micro-kernel has been implemented on a 32-node GP1000 BBN Butterfly, SGI multiprocessor, and a 32-node KSR. Since, currently we are not interested in issues regarding protection, the prototype is implemented completely at the user level.

Operation	BBN	KSR
null function call	3.33	1.78
thread-fork	475.67	71.5
thread-yield	219.96	38.69
pingpong	434.81	78.09
spin-lock	40.79	5.21
blocking-lock	88.59	6.89
monitor-an-attribute	66.03	3.35
reconfigure-an-attribute	9.87	2.06

Table 1: Costs (μseconds) of the basic micro-kernel operations

Table 1 lists the costs of a few basic operations provided by the micro-kernel implemented on two different machines – BBN Butterfly and KSR. The operation *thread-yield* just yields the processor to the next ready thread, the operation `pingpong` bounces the processor between two threads. The numbers listed in the table for spin-locks and blocking locks are obtained from the reconfigurable lock – configured as spin and block. These numbers are comparable to primitive spin and blocking locks. The operations `monitor-an-attribute` and

`reconfigure-an-attribute` are the basic attribute monitoring operation, and attribute reconfiguration operation respectively.

We have already investigated the possible application performance gains due to dynamic reconfiguration of locks[16, 17]. Using artificial work-loads[16] and a representative multiprocessor application, a parallel branch-and-bound program, on a 32 node BBN Butterfly Multiprocessor[14] we demonstrate that:

- For improved performance, an application requires lock schedulers that are most appropriate for the locking patterns exhibited by its threads. An experiment with a client-server application on the BBN Butterfly multiprocessor demonstrates that priority and handoff lock schedulers outperform FIFO lock schedulers by 13%, thus demonstrating the importance of application-specific lock schedulers (*configurability*) [16].

- The optimal waiting policy for a lock depends on the application's dynamically changing locking pattern. An experiment with reconfigurable locks with a workload generator on a BBN Butterfly multiprocessor demonstrates that dynamic reconfiguration of waiting policy considerably enhances application performance (*reconfigurability*)[16].

- Experimentation with the parallel branch and bound program, and Time-Warp kernel demonstrates that "closely coupled" adaptive locks[17] outperform other available locks by up-to 17%, thus demonstrating the need of adaptation in operating system abstractions (*adaptability*).

Repetition of the above experiments with the Travelling Sales Person application, and the Time-Warp kernel on KSR demonstrated similar results[11].

Our current research concerns the demonstration of similar performance gains from the dynamic configuration of thread schedulers. Specifically, for improved performance and ease of programming, we posit that an application should be provided with the scheduler that best suits its needs. We are studying the following issues. First, we are now investigating the runtime configuration of a single thread scheduler – termed scheduler configuration – performing real-time scheduling for Time-Warp discrete event simulations[10]. Second, the performance of many applications can be improved if different, concurrently executable parts of their computations can be scheduled with differing thread schedulers – termed *heterogeneous scheduling.* Currently, we are studying scheduling characteristics of different applications to investigate whether heterogeneous scheduling technique results in application performance improvement. Initial results of the experimentation on dynamic scheduler reconfiguration and heterogeneous scheduling are documented in a companion paper[15].

5 Related Research

Some of the notions introduced in PRESTO[3], CHOICES[6] and CHAOS[9] are similar to our work. PRESTO uses the encapsulation property of objects to build a configurable parallel programming environment. Structurally, PRESTO's synchronization object is somewhat similar to an adaptive lock[17]. However, a synchronization object does not support dynamic reconfiguration of implementation-dependent attributes, adaptation, and object state monitoring. CHOICES is an example of an object based reconfigurable operating system which can be tailored for a particular hardware configuration or for a particular application. The focus of CHOICES is to structure the operating system kernel in an object oriented way whereas the focus of our research is to build a micro-kernel that can be reconfigured at the thread level. Even though we use the object model at the application level, we do not use objects to build the run-time system. CHAOS[2] is a family of object-based real-time operating system kernels. The family is *customizable* in that existing kernel abstractions and functions can be modified easily. CHAOS supports reconfiguration at the object level and is structured on top of the reconfigurable micro-kernel.

6 Conclusion

The major contributions of this paper are twofold:

1. First, it proposes an architecture for a reconfigurable micro-kernel including the introduction of implementation-specific mutable attributes, object attribute monitoring, implementation dependent adaptation policies, and attribute families.

2. Then, it describes the implementation of reconfigurable kernel components like locks and schedulers.

In this paper, we demonstrate the need for reconfiguration at the micro-kernel level by implementing prototype systems on KSR and Butterfly multiprocessors. Currently, we are using the system to experiment with different component specific adaptation policies. In the future, we will investigate the group reconfiguration problem using the concept of attribute family.

This paper does not address issues regarding operating system protection. Currently, we do not assume any hardware protection boundary between user and the kernel. We, however, hypothesize that the concepts will hold in the presence of protection boundary. But, as any other micro-kernelized operating systems, applications will experience performance degradation due to frequent crossing of the protection boundary.

[2] **A Concurrent, Hierarchical, Adaptable Operating System supporting atomic, real-time computations.**

References

[1] Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. The performance implications of thread management alternatives for shared-memory multiprocessors. *IEEE Transactions on Computers*, 38(12):1631–1644, December 1989.

[2] Thomas E. Anderson, Henry M. Levy, Brian N. Bershad, and Edward D. Lazowska and. The interaction of architecture and operating system design. Technical Report 90-08-01, Dept. of Computer Science and Eng., University of Washington, August 1990.

[3] B. Bershad, E. Lazowska, and H. Levy. Presto: A system for object-oriented parallel programming. *Software: Practice and Experience*, 18(8):713–732, August 1988.

[4] D. Black, D. Golub, D. Julin, R. Rashid, R. Draves, R. Dean, A. Forin, J. Barrera, H. Tokuda, G. Malan, and D. Bohman. Microkernel operating system architecture and mach. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 11–30, April 1992.

[5] David. L. Black. *Scheduling and Resource Management Techniques for Multiprocessors.* PhD thesis, School of Computer Science, Carnegie Mellon University, July 1990. Techreport CMU-CS-90-152.

[6] R. Campbell, G. Johnston, and V. Russo. Choices (class hierarchical open interface for custom embedded systems). *Operating Systems Review*, 21(3):9–17, July 1987.

[7] Christian Clemencon, Bodhisattwa Mukherjee, and Karsten Schwan. Distributed shared abstractions (dsa) on large-scale multiprocessors. In *Symposium on Experience with Distributed and Multiprocessor Systems*, September 1993.

[8] Edward F. Gehringer, Daniel P. Siewiorek, and Zary Segall. *Parallel Processing: The Cm* Experience.* Digital Press, Digital Equipment Corporation, 1987.

[9] A. Gheith and K. Schwan. Chaos-arc – kernel support for multi-weight objects, invocations, and atomicity in real-time appl ications. *ACM Trans. on Comp. Sys.*, pages 33–72, April 1993.

[10] Kaushik Ghosh, Richard M. Fujimoto, and Karsten Schwan. A testbed for optimistic execution of real-time simulations. *IEEE Workshop on Parallel and Distributed Real-Time Systems*, April 1993.

[11] Kaushik Ghosh, Bodhisattwa Mukherjee, and Karsten Schwan. Experimentation with configurable, lightweight threads on a ksr multiprocessor. Technical Report GIT-CC-93/37, College of Computing, Georgia Institute of Technology, 1993.

[12] E. Lazowska and M. Squillante. Using processor-cache affinity in shared-memory multiprocessor scheduling. Technical Report 89-06-01, Department of Computer Science and Engineering, University of Washington, June 1989.

[13] Bert Lindgren, Bobby Krupczak, Mostafa Ammar, and Karsten Schwan. Parallel and configurable protocols: Experiences with a prototype and an architectural framework. Technical report, College of Computing, Georgia Institute of Technology, GIT-CC-93/22, Atlanta, GA, March 1993. To appear in 1993 International Conference on Network Protocols.

[14] Bodhisattwa Mukherjee. A portable and reconfigurable threads package. In *Proceedings of Sun User Group Technical Conference*, pages 101–112, June 1991. TR# GIT-ICS-91/02.

[15] Bodhisattwa Mukherjee and Karsten Schwan. Application dependent, heterogenous thread scheduling. Technical Report GIT-CC-93/43, College of Computing, Georgia Institute of Technology, July 1993.

[16] Bodhisattwa Mukherjee and Karsten Schwan. Experiments with a configurable lock for multiprocessors. In *proceedings of International Conference on Parallel Processing*, August 1993. Also available as TR# GIT-CC-93/05.

[17] Bodhisattwa Mukherjee and Karsten Schwan. Improving performance by use of adaptive objects: Experimentation with a configurable multiprocessor thread package. In *proceedings of high performance distributed computing*, July 1993. Also available as TR# GIT-CC-93/17.

[18] J. Ousterhout. Scheduling techniques for concurrent systems. In *Proceedings of Distributed Computing Systems Conference*, pages 22–30, 1982.

[19] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrman, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser. Overview of the chorus distributed operating system. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 39–69, April 1992.

[20] Karsten Schwan, Tom Bihari, Bruce W. Weide, and Gregor Taulbee. High-performance operating system primitives for robotics and real-time control systems. *ACM Transactions on Computer Systems*, 5(3):189–231, Aug. 1987.

[21] Karsten Schwan, Ben Blake, Win Bo, and John Gawkowski. Global data and control in multicomputers: Operating system primitives and experimentation with a parallel branch-and-bound algorithm. *Concurrency: Practice and Experience, Wiley and Sons*, pages 191–218, Dec. 1989.

[22] J.T. Schwarz. Ultracomputers. *ACM Transactions on Programming Languages and Systems*, 2(4):484–543, Oct. 1980.

[23] P. Stenstrom, T. Joe, and A. Gupta. Comparative performance evaluation of cache-coherent numa and coma architectures. In *Proceedings of the 19th Annual International Symposium on Computer Architecture, ACM*, pages 80–91, May 19-21 1992.

[24] M. Stumm, R. Unrau, and O. Krieger. Designing a scalable operating system for shared memory multiprocessors. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 285–303, April 1992.

[25] William A. Wulf, Roy Levin, and Samuel R. Harbison. *Hydra/C.mmp: An Experimental Computer System.* McGraw-Hill Advanced Computer Science Series, 1981.

[26] J. Zahorjan, E. Lazowska, and D. Eager. The effect of scheduling discipline on spin overhead in shared memory multiprocessors. Technical Report 89-07-03, Department of Computer Science and Engineering, University of Washington, May 1989.

Cohabitation and Cooperation of Chorus and MacOS

Christian Bac,
Institut National des Télécommunications,
rue Charles Fourier, 91000 Evry, France.
+ 33 1 60 76 45 39, E-mail: chris@int-evry.fr

Edmond Garnier,
Alcatel Alsthom Recherche,
Route de Nozay, 91460 Marcoussis, France.
+ 33 1 64 49 15 77, E-mail: garnier@aar.alcatel-alsthom.fr

Abstract

This paper describes experimental work on cohabitation and cooperation between a distributed operating system (Chorus[1]) and an event driven operating system (MacOS[2]). Our aims were to exploit the graphical and the musical capabilities of Macintosh hardware and software directly from Chorus applications, while minimizing our efforts in the field of device drivers and hardware interfaces.

The work was carried out in four major stages. The first stage was to port the Chorus kernel on the Macintosh hardware. In the second stage we changed the way Chorus managed the hardware in order to keep the MacOS system alive. Conversely, we modified slightly the way Chorus was booted so as to present it as an application to MacOS. This led us to the third stage, which was to share system events (e.g. hardware interrupts) between the two systems. The Chorus system allows one to have multiple functions connected to an interrupt. This feature was used to connect both an internal Chorus driver and a low level function to an interrupt. The low level function leads to the MacOS interrupt driver. The fourth stage is currently being carried out. It consists in the design and implementation of an interface permitting user level events (as system calls) to cross the borders of the two systems.

This paper describes each stage and draws lessons about system software cohabitation and reusability.

1 Chorus is a registered trademark of Chorus Systèmes.
2 MacOS is a registered trademark of Apple Computers Inc.

1. Introduction

The technology of distributed systems [1] [2] has grown and matured since the early eighties [3]. Today it is used to support applications based on the UNIX[1] operating system interface [4] [5] and also new features such as lightweight processes and Inter Process Communications with message passing. Over the same period personal computers became widely used. These computers offer graphical and musical interfaces, and have partially read only memory based operating systems.

Our goal was to combine the two technologies and see whether they worked well together or not. It seemed particularly important to us to be able to use existing technology, including machines and operating systems, and to integrate them into our base of distributed computers. We decided to use a cooperative approach by allowing the two systems (Chorus and MacOS) to cohabit in memory space and to share the CPU.

Our cooperative approach was slightly different from the Mach and Chorus operating systems. Mach, up to the 2.5 version, was embedded in the UNIX system. It has been rewritten in the 3.0 version to support system servers offering a UNIX interface at user level. An early version of Chorus [6] (Chorus V2), cooperated with the UNIX system, but it was on a multiprocessor machine. Chorus V2 ran on several processors and UNIX ran on one. A Chorus driver was integrated into the UNIX kernel and communicated with the Chorus system by using a common memory space. This approach was abandoned because it required very specific hardware architecture. A subsystem (Chorus/Mix) of servers emulating a UNIX system on top of Chorus was created instead. Both Chorus and Mach achieve binary compatibility at the application level but they use source code of the UNIX operating system in their servers. However we were not interested in another UNIX like operating system but wished to add more graphical capabilities, as in the V kernel [7], to the Chorus distributed system. Similar experiments have been done over Mach 3.0 [8] with the DOS or MacOS running as an application.

2. Chorus & Mac project

The Chorus & Mac project began three years ago with the port of a Chorus simulator in the AU/X[2] environment. The Chorus simulator is composed of a UNIX process and a library that can be used to program and test Chorus applications in the UNIX environment. This port was done to gain experience in the Chorus area. This was slightly more difficult than expected, mostly because the simulator used certain BSD features and made assumptions about memory allocations that were different in AU/X (which is System V based). During this port, we explored many parts of the source code, common to the simulator and the Chorus kernel, and debugged C++ code with simple debuggers like `adb` and `sdb` (which obviously were not suitable). The simulator was later used to develop graphical applications using the distributed IPC of the Chorus system in an attempt to anticipate future research.

We then began the major work; porting a Chorus kernel on the Macintosh hardware and trying to exploit MacOS features directly from Chorus applications. This work was done in four major stages. The first stage was to port the Chorus kernel on the Macintosh hardware, the second was to have both systems cohabit but ignore each other and in the third stage we changed the interpretation of the interrupts slightly to accord the Chorus system to the MacOS use of interrupts. This permitted both systems to share system events. The last stage, that is currently being carried out, will allow user applications events to cross the border between the two systems.

---

[1] UNIX is a registered trademark of AT&T USL.
[2] AU/X is the UNIX system on MacIntosh and a registered trademark of Apple Computers Inc..

In the following sections, we will describe, how each stage has been successfully achieved.

2.1. Porting Chorus on Mac Hardware

The target machine is a Macintosh II CX. It is based on a MC68030 and many added chips. We used the Chorus sources for a Tadpole TP33M that is also based on the MC68030 chip.

2.1.1. Chorus kernel design

As shown in Figure 1 below, the Chorus kernel [9] is composed of four independent elements:

- The supervisor dispatches interrupts, traps and exceptions delivered by the hardware.
- The real-time executive controls the allocation of the processor.
- The virtual memory manager is responsible for manipulating virtual memory hardware and local memory resources.
- The inter-process communication manager provides message passing facilities.

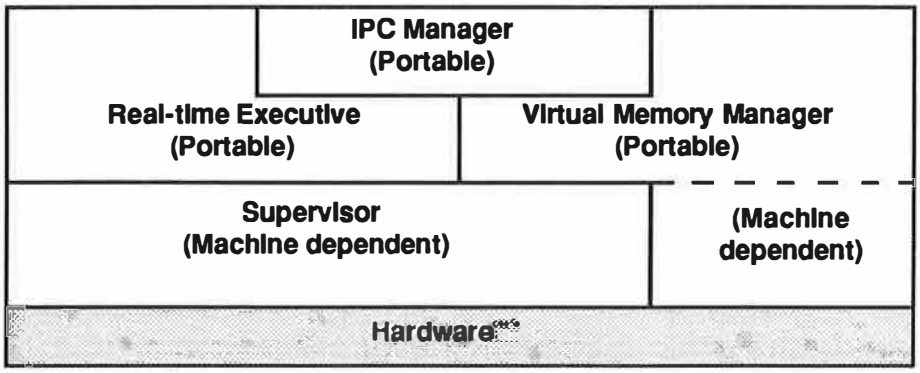


Figure 1 - *Chorus Kernel Architecture*

Thus most of the system is portable. This part was not modified because we used the sources for the same kind of architecture (MC68030).

2.1.2. Macintosh hardware

As shown in Figure 2 below, the Macintosh II CX hardware [10] that was used, is composed of:

- a Motorola MC68030 microprocessor which integrates a Paged Memory Management Unit equivalent to the MC68851,
- a Motorola MC68882 floating point unit,
- eight megabytes of RAM,
- 256 KB of ROM which contains a large part of the MacOS software,
- six extension slots using the NuBus standard: one is used for the Ethernet board and one supports the video board, the NuBus is interfaced to the main processor bus by logic circuits referred as Bus Interface Units (BIUs),
- regular chips controlling hardware:
  - a Zilog Z8350 serial communication controller (SCC) providing two ports for serial communications,
  - an NCR 5380 Small Computer System Interface (SCSI) controlling an internal 80 MB hard disk and the external SCSI connector,

– and Apple custom chips used in others Macintosh:
  - the GLUE, which provides address decoding and controls signals,
  - the Apple Desktop Bus (ADB), which handles communications with the keyboard and the mouse,
  - two Versatile Interface Adapters (VIA), based on the Rockwell 6522 chip, which support the ADB, the Real Time Clock, timers and other I/O devices,
  - a custom digital sound synthesizer chip, called Apple Sound Chip (ASC),
  - and the IWM (Integrated Woz Machine), which controls the floppy disk.

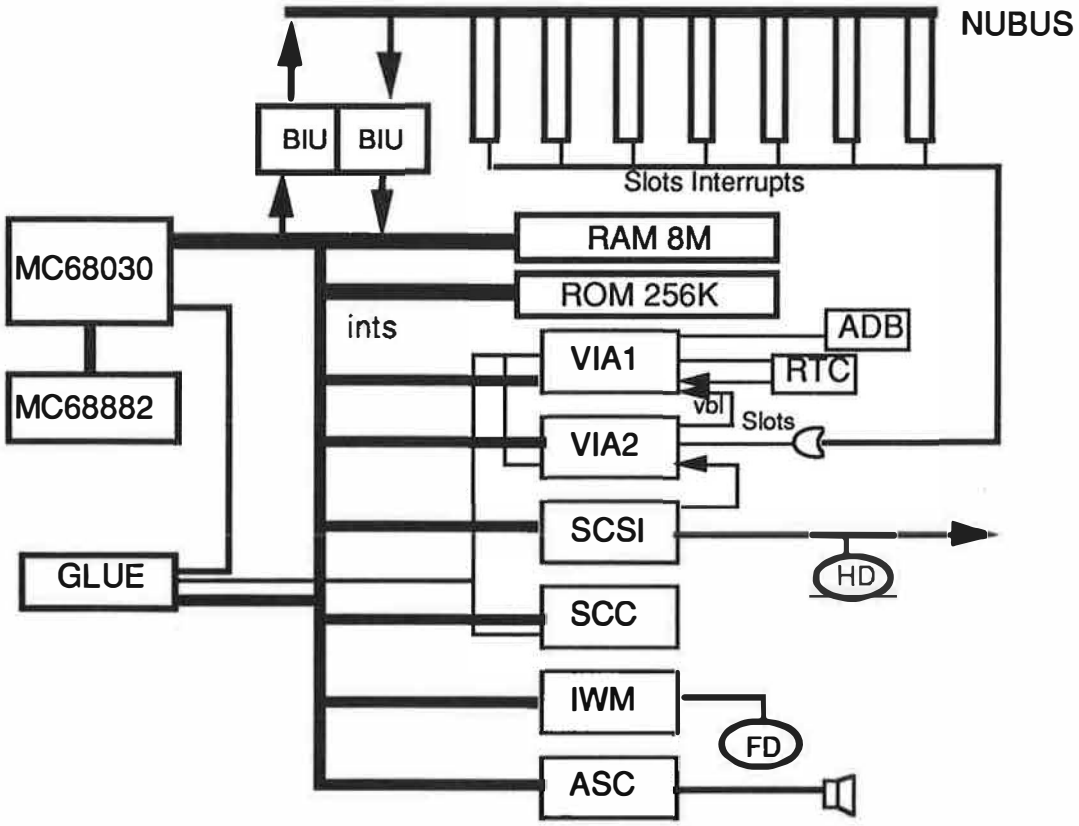


Figure 2 - *Macintosh II hardware*

2.1.3. Chorus hardware requirements

Chorus hardware requirements are light. To run, the kernel needs a timer that performs an interrupt every 1/100th second, and a terminal interface that reads and writes characters. It also requires correct initialization of the memory management unit, we will discuss this point later on. Timing has been achieved by using timer one of the second VIA. This timer is used in MacOS to generate a signal known as the VBL (Vertical Blanking Interrupt). In MacOS, the handler associated with this signal performs periodic events such as blinking or reading mouse position, etc. We decided to connect the terminal to port A of the SCC. This port is usually used to pilot a modem in MacOS or a terminal in AU/X. The low level routines allowing Chorus to control the port usage were written in C.

2.1.4. The development environment

We were working in a cross development environment. We used a SUN 3 to compile and build the Chorus archive. This file contains a boot program, the Chorus kernel, and a minimal set of actors. We transferred this archive to the Macintosh using AU/X and TCP/IP, then we rebooted and used the Standalone Shell

(SASH). SASH is a MacOS application which gives access to the UNIX partition. SASH is capable of starting applications that have been developed in the AU/X environment and linked to the SASH library. These applications are capable of using many UNIX system calls that are translated by the SASH library. These applications also have access to the UNIX disk partition.

We developed a preboot program that was launched by SASH. This preboot program loaded the Chorus archive in the memory and gave control to the boot program (inside the Chorus archive). The boot program tested the memory size and set up the PMMU translation tables. The boot program then started the Chorus kernel.

2.1.5. Translation tables

In general, the management of the MMU must be written; in our case the greater part had already been done, due to the sources used. The initialization of the translation tables remained the main problem. The physical address space of the Mac is described in Figure 3 below.

F100 0000 - FFFF FFFF	Standard NuBus space
F000 0000 - F0FF FFFF	Reserved
6000 0000 - EFFF FFFF	NuBus Slot expansion space
5000 0000 - 5FFF FFFF	I/O space
4000 0000 - 4FFF FFFF	ROM
0000 0000 - 3FFF FFFF	RAM

Figure 3 - *Physical address space on the Mac II*

Chorus on MC68030 uses a virtual address space composed of 4 K Bytes memory pages and four levels of memory index as shown in Figure 4.

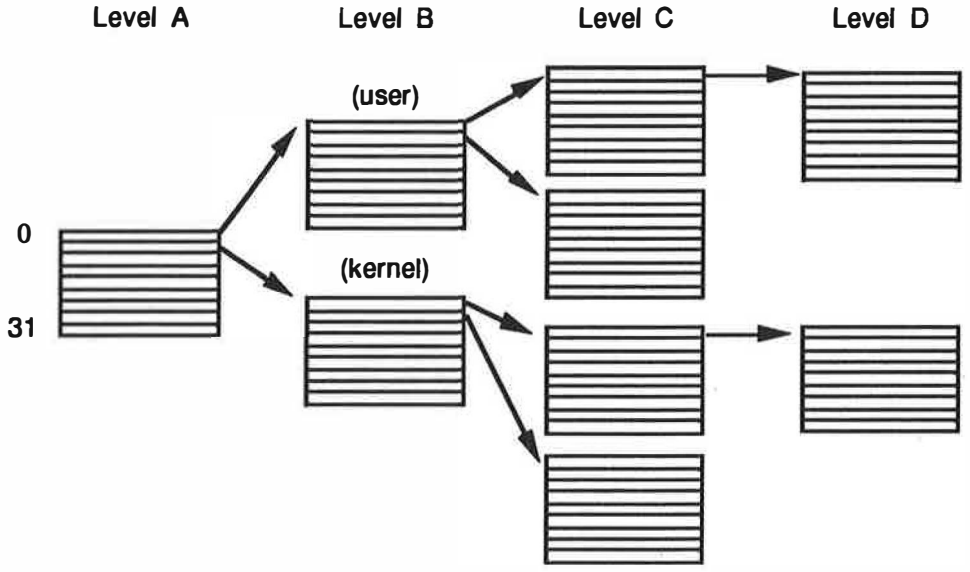


Figure 4 - *Translation tables for Chorus*

The translation tables are set up at boot time by the boot program. When Chorus is running, the first entry describes the actor running in the user space, and the second entry describes the Chorus kernel space. This space includes the actors running in the supervisor or system mode. A user actor is allowed to use the address range from 0x0000 0000 to 0x07FF FFFF. The Chorus kernel uses the address range from 0x0800 0000 to 0x0FFF FFFF for its private space. In the Mac version, the other level A entries are reserved to the kernel and map the physical address space allowing the kernel to manipulate circuits like the VIA or the SCC, in the address range 0x1000 0000 to 0xFFFF FFFF.

2.1.6. Conclusions of stage one

The use of the Macintosh hardware by Chorus was limited to SCC and VIA. The SCC was used to transmit and receive characters. The first timer of the second VIA was programmed to emit clock ticks. Although our development environment was complicated, we succeeded in completing each stage necessary to bring Chorus into the Macintosh memory. Despite the complexity of the Macintosh hardware, Chorus has proved to be easily portable on it. In fact, we were lucky to have sources for MC68030. Stage one was completed and we had learned a lot about Chorus memory management and Macintosh hardware. We then tested the kernel with the "Chorus kernel tests". They ran successfully and we let them run more than one week with no error.

At this stage the Chorus system had only been modified to support the hardware and no attention had been paid to the MacOS system. This was to be our next stage.

2.2. Bringing MacOS back to life

In the second stage we changed the way Chorus managed the hardware to keep the MacOS system alive. Conversely, we slightly modified the way Chorus was booted so as to present it as an application to MacOS.

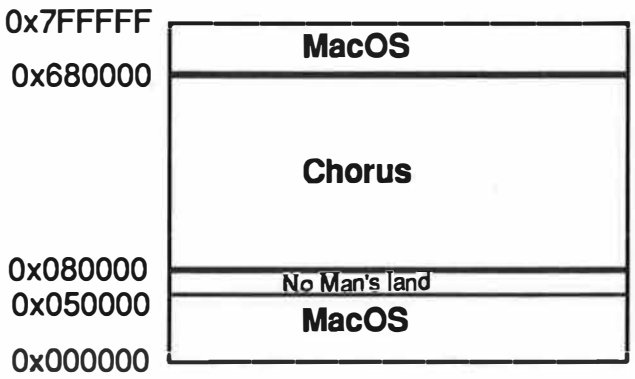


Figure 5 - *Sharing memory space*

2.2.1. Sharing memory space

To favour cohabitation of Chorus and MacOS, the physical memory is separated in three areas (see Figure 5 above):

- the memory reserved for the MacOS system (composed of two physical parts),
- the memory reserved for the Chorus system (where the kernel and actors are placed),

- and a third area that initially allowed a correct alignment of Chorus memory space and that was later used to go from one system to the other, and that we will call "No Man's Land".

We decided to limit the physical memory reserved for Chorus to 6 MB. Chorus can behave as if memory does not begin at physical address 0. After tests on memory allocation in MacOS, we chose to start the memory space reserved for Chorus at the physical address 0x80000. To share memory space with MacOS, we asked for memory allocation to it during the preboot phase. This was done by rewriting the preboot program.

2.2.2. New Chorus preboot

Thus a new Chorus preboot was written. It reserved the 6.5 MB memory block at MacOS level. It installed the Chorus archive in memory, and it allowed a direct boot from MacOS without using the SASH. As explained below, this program was further extended to act in MacOS space for Chorus. This brought the MacOS system partly back to life. We were thus able to stop Chorus and return back to MacOS as in a Mac application. The first stage in collaboration had been achieved successfully and the two systems could share memory. The next stage was the sharing of system events.

2.3. Sharing system events

Now that we had both systems in memory and Chorus running all the time, we wanted to give more control to MacOS. This was achieved by giving control to it when system events appeared. To allow system events to be treated either by MacOS or by Chorus, we needed some low level functions that enabled the CPU to commute from one system space to the other.

2.3.1. Crossing the frontier

When it is running, MacOS uses a 24 bit address space. This address space is mapped to the physical address space as shown in Figure 6 below. The lower part of the address space is used for physical memory.

F0 0000 - FF FFFF	50F0 0000 - 50FF FFFF	I/O space
E0 0000 - EF FFFF	FE00 0000 - FE0F FFFF	Standard NuBus space
D0 0000 - DF FFFF	FD00 0000 - FD0F FFFF	
C0 0000 - CF FFFF	FC00 0000 - FC0F FFFF	
B0 0000 - BF FFFF	FB00 0000 - FB0F FFFF	
A0 0000 - AF FFFF	FA00 0000 - FA0F FFFF	
90 0000 - 9F FFFF	F900 0000 - F90F FFFF	
80 0000 - 8F FFFF	4080 0000 - 408F FFFF	ROM
00 0000 - 7F FFFF	0000 0000 - 007F FFFF	RAM

Figure 6 - *Virtual to Physical address space under MacOS*

The memory in address range from 0X50000 to 0X80000 is used to place assembly functions that are called in order to commute from one system context to the other. Most of the work done in the context switches consists in changing the PMMU setting and the value of the interrupt base register. These modifications allow the Macintosh to act in both memory modes, either 24 bits when in MacOS or 32 bits virtual in

Chorus. The assembly functions are placed in memory by an enhanced preboot program. They use a part of this memory space to save and restore the context registers in memory locations.

There are three functions:

- a boot function that saves the MacOS system context at boot time, before starting the Chorus system,
- a Chorus to MacOS function that saves the Chorus system context and restores the MacOS context,
- and a MacOS to Chorus function that saves the MacOS context and restores the Chorus context.

The contexts are constituted by all the MC68030 registers plus the PMMU registers that point to the translation tables. To save or restore the contexts, an intermediate memory mapping is set up, which allows access to the physical memory and system kernel space.

2.3.2. Interrupts

Interrupts were a kind of nightmare; at first glance we could not split the circuit usage between the two systems. We had to consider a new distribution of events because any event could appear in both systems. We knew what to do when MacOS interrupts arrived in Chorus space and we were able to relay the interrupt to MacOS, but we had no control over what was happening in MacOS. We decided to let MacOS take care of interrupts and just used one of them to trigger a vital part of Chorus, namely, the scheduler.

This was possible at the Chorus level because we could connect an interrupt to multiple tasks. This means that we could connect the timer interrupt to the Chorus function *timein()* and to the jump in MacOS if necessary that leads to the MacOS interrupt handler. The Chorus function *timein()* counts clock ticks and schedules user tasks. The jump in MacOS was achieved by connecting the interrupt level to a function in "No man's land" that simulated the interrupt on the MacOS stack and jump to the MacOS interrupt handler. The interrupt frame must be duplicated from the Chorus interrupt stack to the MacOS stack. As MacOS was not able to acknowledge the timer interrupt correctly, the interrupt generated by the timer was blocked in the VIA before the jump in MacOS. The interrupt was released on the return from MacOS. We checked that very few ticks were lost over a long period by using this method.

2.3.3. Events in MacOS

The jump into MacOS allowed it to acknowledge the interrupts correctly. However we wanted MacOS to do more. As mentioned previously, MacOS is an event driven system, and in any MacOS application there is an event loop that handles the keyboard event, and the pull down event, etc. We decided to call a function in our preboot program (MacOS space) to perform certain events on the VBL interrupt. This routine was called from the function in "No man's land" that was associated with the VBL interrupt.

This brought MacOS back to life in many operations:

- mouse moving was detected and the mouse pointer was displayed on the screen,
- accessories like Clock and Super Clock displayed the correct time and changed every second,
- the mouse position could be read from the program,
- and keyboard events were recognized.

2.3.4. What events should be allowed?

We were able to use MacOS features by adding them to the boot program that acted in the Chorus name in MacOS space. For example, we added a function that was triggered by the click in the close box. This function initiated the Macintosh sound manager so that it played music for 30 seconds. Once initiated, the sound manager returned and Chorus continued. The buffer of the ASC is automatically loaded with the bytes representing the sounds to play when the VBL interrupt occurs. The music was correctly played by the

MacOS sound manager although Chorus was running. One question still remained: What should we do with long events? In MacOS, some functions wait for the user to terminate an action before giving the control back. For example, the function allowing the dragging of a window with the mouse or any event involving a long click on the mouse, does not give the control back until the mouse button is released. We decided to disallow these functions because they delayed the Chorus part of the system too much.

At this point, the two systems were present in memory and they could share system events like interrupts. The next stage was to allow user events to be shared by the two systems.

2.4. Sharing user events

We are currently working on the sharing of user events by the two systems. Our aim is to allow user tasks in Chorus to call functions in MacOS. To do this, there are two major problems: the system call must be recognized by Chorus as a MacOS one, and the parameters and results of the system call must be exchanged between the user task in Chorus, and the MacOS system.

2.4.1. Heterogeneous system calls

The user level events or system calls are achieved differently in the two systems. In Chorus, a system call is made via a trap and the interpretation of certain CPU register values gives the system call to the kernel. In MacOS, system calls are implemented as illegal instructions and the system call number is included in the instruction. This allows a great number of system calls. When an illegal instruction can be interpreted as a MacOS system call in the Chorus space, the Chorus kernel must call the function associated in MacOS.

2.4.2. Parameters and results

A system call is generally associated with arguments. In a classic operating system (like UNIX), the arguments are copied from user space to system space before the kernel function is called. When the function has been completed, the results are passed from the system space to the user space. In the case of system calls from Chorus actors to MacOS, the arguments must be copied from an actor user space to MacOS, and back from MacOS to the actor memory space.

This leads to general problems in passing arguments and getting results from functions executed in different memory spaces. This kind of problem is encountered in remote procedure calls [11] and the first solution is to copy the arguments and the results from one space to the other. It seems that it is not the best solution for systems acting in close memory space, and before doing anything, we preferred to explore two different fields in this area:

- a taxonomy of user to system functions calls and usage of parameters and results in these functions,
- measurements of the time spent in system commutation and the construction of arguments and results.

3. Experience and future work

We were able to use the Chorus kernel features to make it cohabit with MacOS. During this experience we have learnt much about Chorus and MacOS. We have proved that both systems were well designed. In fact, they are so well designed that they can be used to do things for which they were not initially built.

In our most recent version, we make both systems co-operate so that Chorus uses MacOS to read from the keyboard and write to the screen. To achieve this co-operation, we need an application in the MacOS system space acting on behalf of the Chorus system. This application is responsible for the correct reservation of resources and for calling the MacOS system. The exchange of data between Chorus and MacOS is done by

using a Macintosh feature in memory space addressing. The Macintosh gives a cyclic vision of its memory that allows the same physical memory locations to be accessed with different addresses from system space in Chorus and from MacOS. For example the physical memory address 0x60000 (corresponding to a physical location in the "No Man's land") is used as a buffer to exchange the characters between Chorus and the preboot application. The buffer can be accessed in Chorus at the address 0x10060000. This virtual address is mapped to the same physical address by the MMU, but the cyclic vision of physical memory given by the Macintosh results in the physical memory at the address 0x60000 to be accessed instead. Thus the exchange of data was done with no modification in the setting of the PMMU.

The Chorus kernel on Macintosh appears robust. We have tried our system with the new version of MacOS (system 7) and it still works. Maybe cohabitation can be improved since the system context switch is slow. The two systems co-operate but they are still not mixed enough. We think that some kind of sub-system, like Chorus MIX, acting as MacOS would be faster and better designed.

The problem of access to resources usage at the user level, is still not completely solved and some further work is required to have an efficient use of cohabiting system resources.

4. Conclusion

We have demonstrated that system cohabitation is possible and that it is a convenient way to extend the functions of a system. The cohabitation results in many advantages compared to the usage of the ROM routines from Chorus:

- by using the MacOS system in RAM, the system benefits from routines that are not in ROM. New versions of some ROM routines and bugs corrections are also accessible.
- the system also uses the interrupt handler from MacOS. The handlers acknowledge the hardware interrupt and post the MacOS events in the event queues in RAM.
- further releases of the MacOS operating system could be used in cohabitation. This would allow an evolving operating system which take advantage of future developments in MacOS software.

Although this work is very specific, the idea of mixing technologies from different areas in the same computer at the lower level is both interesting and advantageous. Cohabitation will allow Chorus real time processes to act in an event driven environment while using a well-known graphical interface with very few developments at the system level. The Chorus operating system can use the MacOS device drivers to manage hardware. The Chorus operating system thus acts as a MacOS application to access peripherals through MacOS.

References

[1] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard and W. Neuhauser. Chorus distributed Operating systems. *Computing Systems*, 1(4), 1988.

[2] Accetta, M.J., Baron, R.V., Bolosky, W., Golub, D.B., Rashid, R.F., Tevanian, A., and Young, M.W. Mach: A New Kernel Foundation for UNIX development. In Proc Usenix Summer, July, 1986.

[3] H. Zimmermann, J.S. Banino, A. Caristan, M. Guillemont, and G. Morisset. Basic concepts for the Support of Distributed Systems: the Chorus approach. In*Proc. 2nd IEEE Int. Conf. on Distributed Computing Systems,* Versailles (France), April 1981.

[4] F. Hermann, F. Armand, M. Rozier, M. Gien, P. Léonard, S. Langlois and W. Neuhauser. Chorus, a new technologie for building UNIX systems. In *Proc. EUUG Autumn '88 Conference*, Cascais (Portugal) 1988.

[5] *Guide to OSF/1: A Technical Synopsis.* O'Reilly & Associates, Inc. 1991.

[6] F. Armand, M. Gien, M. Guillemont, and P. Léonard. Toward a Distributed UNIX system: The Chorus approach. In *Proc. EUUG Autumn '86 Conference*, Manchester (UK) 1986.

[7] E.J. Berglund, An introduction to the V-System, IEEE Micro vol. 6, no 4, August 1986.

[8] G. Malan, R. Rashid, D. Golub, and R. Baron, DOS as a Mach 3.0 Application.

[9] Chorus Systèmes, Overview of Chorus Distributed Operating Systems, from Chorus v.3 Programmer's reference Manual.

[10] *Apple MacIntosh Family Hardware Reference*, Addison Wesley ref. 19255, 1988.

[11] Sun Microsystems, Inc., "RPC: Remote Procedure Call Protocol specification: Version 2", RFC 1057, 1988.

Kernel Support for the Wisconsin Wind Tunnel*

Steven K. Reinhardt, Babak Falsafi, and David A. Wood
Computer Sciences Department
University of Wisconsin–Madison
1210 West Dayton Street
Madison, WI 53706 USA
wwt@cs.wisc.edu

Abstract

This paper describes a kernel interface that provides an untrusted user-level process (an *executive*) with protected access to memory management functions, including the ability to create, manipulate, and execute within subservient contexts (address spaces). *Page motion callbacks* not only give the executive limited control over physical memory management, but also shift certain responsibilities out of the kernel, greatly reducing kernel state and complexity.

The *executive interface* was motivated by the requirements of the Wisconsin Wind Tunnel (WWT), a system for evaluating cache-coherent shared-memory parallel architectures. WWT uses the executive interface to implement a fine-grain user-level extension of Li's shared virtual memory on a Thinking Machines CM-5, a message-passing multicomputer. However, the interface is sufficiently general that an executive could act as a multiprogrammed operating system, exporting an alternative interface to the threads running in its subservient contexts.

The executive interface is currently implemented as an extension to CMOST, the standard operating system for the CM-5. In CMOST, policy decisions are made on a central, distinct control processor (CP) and broadcast to the processing nodes (PNs). The PNs execute a minimal kernel sufficient only to implement the CP's policy. While this structure efficiently supports some parallel application models, the lack of autonomy on the PNs restricts its generality. Adding the executive interface provides limited autonomy to the PNs, creating a structure that supports multiple models of application parallelism. This structure, with autonomy on top of centralization, is in stark contrast to most microkernel-based parallel operating systems in which the nodes are fundamentally autonomous.

1 Introduction

This paper describes the kernel interface designed to support the Wisconsin Wind Tunnel (WWT) [13], a system for parallel simulation of parallel computers. WWT currently runs on the Thinking Machines CM-5 (a message-passing machine) and simulates cache-coherent shared-memory multiprocessors. Shared memory applications execute directly on the CM-5

*This work is supported in part by NSF PYI Award CCR-9157366, NSF Grant MIP-9225097, a Wisconsin Alumni Research Foundation Fellowship, an A.T.&T. Bell Laboratories Ph.D. Fellowship, and donations from Xerox Corporation, Thinking Machines Corporation, and Digital Equipment Corporation. Our Thinking Machines CM-5 was purchased through NSF Institutional Infrastructure Grant No. CDA-9024618 with matching funding from the Univ. of Wisconsin Graduate School.

node processors, with WWT simulating references to remote data. Shared memory functionality is provided using a fine-grain user-level extension of Li's shared virtual memory [10], as described in Section 2.2. WWT uses a separate address space for each simulated (target) node and services all of its exceptions (e.g., MMU faults) and system call requests (e.g., file I/O). In order to study machines larger than the host, several target nodes timeshare a single CM-5 node. In many ways, WWT behaves like an operating system for shared-memory applications. Alternatively, WWT can be thought as providing a virtual machine abstraction—with a shared-memory MIMD machine atop a message-passing pseudo-SIMD machine—for these applications [8].

WWT requires several unusual features from the underlying operating system. Specifically, the kernel must allow WWT to:

- Create subservient contexts (address spaces)
- Manipulate page mappings within sub-contexts
- Initiate execution in sub-contexts
- Handle traps generated during execution in sub-contexts
- Manage physical memory tags in sub-contexts[1]

All of these features must coexist with traditional memory management functions, including paging and/or swapping.

We have defined and implemented an interface which provides these features and call any application that makes use of them an *executive*. While the interface is motivated by our specific application, it provides any untrusted user-level process with protected access to memory management functions, including the ability to create, manipulate, and execute within subservient contexts (address spaces). *Page motion callbacks* not only give the executive limited control over physical memory management, but also shift certain responsibilities out of the kernel, greatly reducing kernel state and complexity.

Because an executive creates contexts and controls them completely, it can act as the operating system for other applications, providing an execution model not available under the native operating system. For example, an executive can export various thread and memory abstractions without adding complexity to the kernel itself.

While this flexibility is useful in a uniprocessor context, it is particularly important on the CM-5, since standard CMOST is a centralized, synchronous operating system, allowing little autonomy for the node kernels. CMOST's structure efficiently supports an important class of parallel applications, i.e. fine-grain data-parallel codes, but cannot take advantage of more autonomous execution models. By extending CMOST with the executive interface, we provide a kernel structure that can efficiently support both synchronous and asynchronous applications.

The combination of CMOST and executive interface results in a unique kernel structure that provides autonomy on top of synchrony, rather than the more traditional approach of coordinating fundamentally autonomous nodes. This new kernel structure may prove superior because centralized control appears to have advantages for supporting fine-grain synchronous codes and managing global hardware resources, while the executive interface provides flexible support for other execution models.

[1]We have effectively extended the CM-5 architecture to support two bits of tag information for each 32-byte block of physical memory; see Section 2.3.

The next section provides background on the Thinking Machines CM-5 system, the Wisconsin Wind Tunnel, and our method of synthesizing memory tags on the CM-5. While the interface is not tied to any of these, this section provides context and motivation for the rest of the paper. Section 3 defines the interface, consisting of context manipulation calls, page motion callbacks, and execution management calls. Section 4 describes our implementation of the interface in CMOST and its performance. Section 5 discusses the implications of this work for the structure of multiprocessor operating systems. Finally, we discuss related work and our conclusions.

2 Background

2.1 Thinking Machines CM-5 and CMOST

The Thinking Machines CM-5 [16] is a distributed-memory message-passing multiprocessor. Each processing node consists of a 33 MHz SPARC microprocessor with a cache and memory management unit, up to 128 MB of memory, a custom network interface chip, and optional custom vector units. The processing nodes are grouped into partitions of 32 or more processors. Each partition is managed by a control processor (CP), distinct from the processing nodes (PNs).

The standard operating system for the CM-5 is CMOST. Under CMOST, all policy decisions, including scheduling, swapping, and memory allocation, are made on the control processor. The processing nodes execute a minimal microkernel (the *PN kernel*) which provides the bare mechanisms required to implement the CP's policy. Because all processors in the partition are managed as a synchronous unit, CMOST gives the CM-5 some SIMD-like qualities. For example, when the CP decides to context switch, all nodes simultaneously switch to the new context. Similarly, when a new process is created, the CP selects the physical pages which the process will occupy on the PNs and broadcasts that process's memory map.

CMOST and the CM-5 are optimized to run data-parallel applications, where all nodes synchronously apply similar operations to a local subset of a global data structure. In particular, the CM-5 contains a "control network", distinct from the message-passing network, which provides hardware support for global operations such as barriers, reductions, and broadcasts [9]. To efficiently utilize this control network, all nodes in a partition must concurrently execute the same user process. The centralized CMOST structure automatically satisfies this condition.

2.2 The Wisconsin Wind Tunnel

The Wisconsin Wind Tunnel (WWT) provides a platform for evaluating parallel computer systems—specifically cache-coherent shared-memory computers—by accurately modeling the performance of real workloads on proposed hardware [13]. WWT helps computer engineers evaluate computer architectures much like a wind tunnel helps aeronautical engineers design aircraft. WWT uses the execution of a parallel shared-memory application to drive a distributed discrete-event simulation, accurately calculating the execution time of that application on a modeled hardware system (the *target*). Events generated by the simulation, such as lock acquisitions and memory reference completions, are used in scheduling the application, guaranteeing that the application's execution proceeds exactly as it would on the target system.

We call WWT a *virtual prototype* because it uses direct execution to leverage similarities between the target system and the system on which it executes (the *host*) [5]. This means that the target application executes directly on the host hardware as much as possible—for example, a target floating-point multiply runs as a host floating-point multiply. Software simulation is required only for those features of the target system not provided by the host.

Because WWT executes on a message-passing machine, the primary feature it must simulate is the shared memory abstraction. We do this using a fine-grain extension of Li's shared virtual memory [10]. Shared virtual memory constructs a distributed shared memory using standard address translation hardware to control memory access on each node. If a node has a copy of a shared data page, it is mapped into the address space on that node; if a node has no copy, the virtual page is not mapped. Multiple read-only copies are easily supported using the page protection facilities. Program accesses that require a data transfer to acquire a valid or exclusive copy are signaled as page faults. Unfortunately, relying on address translation hardware alone restricts the granularity of coherence to at least the virtual memory page size.

The shared-memory machines we wish to model maintain coherence at a finer granularity, typically tens of bytes rather than thousands. We have synthesized the ability to tag each 32-byte block in physical memory as *invalid*, *read-only*, or *writable* (see Section 2.3). Using these tags in combination with the address translation hardware, we implement a distributed shared memory that maintains coherence at a 32-byte granularity. The first reference to a shared page causes a page fault, as with shared virtual memory. We allocate and map a physical page, but initially mark each cache block invalid. Cache blocks are marked valid (read-only or writable) only as they are referenced. Accesses to invalid blocks (and writes to read-only blocks) cause faults, and initiate software that fetches the data and marks the block valid. The distinction between read-only and writable tags allows read replication at the cache block granularity.

WWT uses this fine-grain shared virtual memory to directly execute shared-memory applications as they would execute on a cache-coherent target machine. A context is allocated for each target node; the shared data accessible from this context reflects the contents of the simulated cache on that target node. Page and tag faults correspond to target cache misses, which invoke WWT and are handled according to the target's coherence protocol. Large target systems are studied by allocating several contexts per host node and multiplexing their execution.

2.3 Memory tags

To implement fine-grain shared virtual memory, blocks in memory must have three states: invalid, read-only, and writable. Any access to an invalid block and write accesses to read-only blocks must provide restartable exceptions. To achieve this functionality, we have logically extended the CM-5 architecture to support two additional bits of information—*writable* and *invalid*—per 32-byte physical memory block.

Although memory tags with access semantics have appeared in numerous machines, e.g., the Denelcor HEP [15], most contemporary commercial machines—including the CM-5—do not provide this capability. However, we are able to synthesize an invalid tag on the CM-5 by forcing uncorrectable errors in the memory's error correcting code (ECC) via a diagnostic mode. Using the SPARC cache in write-back mode causes all SPARC cache misses to appear to the memory as cache block fills. A fill that encounters an uncorrectable ECC error generates a precise exception.

Synthesizing a read-only state is more convoluted, since it requires using the page tables to make entire pages read-only. On a write fault, we must distinguish between a write to a read-only block and a write to a writable block that resides on the same page as one or more read-only blocks. We make this distinction by maintaining a bit vector—one bit per block—to indicate whether the block is writable. The write fault handler checks this bit; if set it performs the write and resumes the application, rather than signaling a fault.

Memory tags introduce extra state—two additional bits per 32-byte block—making paging and swapping more complex to implement. The tag bits make the "extended" physical page no longer a power of two, causing a mismatch with typical disk block sizes, and requiring more bookkeeping and I/O operations. In addition, since memory tags are unused for many pages, e.g., text and non-shared data, any overhead maintaining them is wasted. The executive interface reduces the kernel's complexity by shifting responsibility for maintaining memory tags to the executive.

3 The executive interface

The executive interface provides an *executive*—an untrusted user-level process—with protected access to memory management functions. An executive can use the interface's memory management calls to create subservient contexts (address spaces), and exert complete control over them, including adding, modifying, and deleting page-level mappings. An executive can invoke execution within a subcontext, and regain control on all faults and exceptions. Page motion callbacks not only give the executive limited control over physical memory management, but also shift certain responsibilities out of the kernel, greatly reducing kernel state and complexity.

A primary goal of the executive interface is to minimize kernel state and complexity. Beside the aesthetic appeal, keeping most of the code and complexity at user level makes bugs less catastrophic and easier to eliminate. In addition, because we are modifying a continuously developing system, minimizing and isolating the kernel source changes makes it easier for us to keep up with vendor revisions.

A particular challenge is maintaining the address mappings and memory tag values installed by an executive in the face of paging/swapping activity by the kernel. The brute-force solution is to have the kernel remember all of the mapping requests made by the executive and transparently maintain them when a page is swapped out and back in at a different physical address, and to transparently swap tag information as well as page data. We have solved this problem with greatly reduced kernel state and complexity using *page motion callbacks*. These callbacks allow the kernel to notify an executive immediately before a page is to be swapped out and immediately after it is swapped back in so that the executive itself can maintain address mappings and tag values.

Another requirement is to avoid trusting the executive. A fully protected interface makes the system robust through even the earliest phases of executive development, and means that normal users have the ability to write or modify executives. A protected interface also makes other sites more willing to adopt our kernel so that we can distribute the Wisconsin Wind Tunnel. Two features of the interface contribute to this protection:

- The executive cannot even refer to resources not explicitly allocated to it by the kernel. The executive never sees physical addresses or hardware context numbers.
- The kernel guarantees that the executive never has an alias to a physical page it does

```
int create_ctx();
void *executive_brk(void *new_brk);
void *executive_sbrk(int incr);
void *executive_vbrk(void *new_brk);
void *executive_vsbrk(int incr);
int add_mapping(int cd, void *va, void *pp, int attr);
int change_pg_attr(int cd, void *va, int attr);
int delete_mapping(int cd, void *va);
void jump_to_ctx(int cd, struct regs *p, void *stackp);
```

(a) General kernel calls

```
int set_page_motion_cbs(void *(*page_going)(),
                        void (*page_coming)(),
                        void *stackp);
int set_ctx_fault_cb(void (*ctx_fault_cb)());
```

(b) Callback registration calls

```
void *page_going(void *pp);
void page_coming(void *pp);
void ctx_fault(int fault_code, struct regs *p, ...);
```

(c) Callbacks

Table 1: The executive interface. Parts (a) and (b) list the functions exported by the kernel. Part (c) describes the callbacks exported by the executive.

not own by maintaining a count of aliases for each physical page. If the executive does not decrement this count to zero by deleting mappings before a page is removed from its control, the kernel will terminate it.

Table 1 lists the calls and callbacks that comprise the executive interface. The kernel exports the general calls and callback registration functions, while the executive exports the callbacks, which it registers during initialization. The bulk of the interface is directly related to managing virtual and physical memory. The remaining functions, jump_to_ctx() the ctx_fault() callback, provide the ability to execute within subcontexts.

3.1 Memory management

The executive manages virtual and physical memory resources via the context management calls and page motion callbacks. Pages managed by the executive are distinct from the executive's own text, data, and stack pages. This allows the kernel to easily distinguish the former for special handling while manipulating the latter as it would for any other user process. For example, the kernel can swap the executive's text segment, share it among multiple instances of the same executive, or allow a debugger to attach to an executive without interfering with the executive's memory management functions.

3.1.1 Context management calls

The context management calls allow an executive to create new contexts, allocate pages to map into them, and add, modify, and delete page-level mappings. Using these calls, an executive has complete control over these subservient address spaces.

The create_ctx() call allocates a new context and returns an integer context descriptor (similar to a Unix file descriptor). We refer to these contexts as *subcontexts* when it is necessary to distinguish them from the executive's context. A new subcontext is completely empty, i.e. it contains no valid address mappings (except for the kernel mappings required by the SPARC architecture). Context descriptor 0 is never returned by create_ctx() and is used to indicate the executive's own address space. The notation cd:va refers to virtual address va in context cd.

To keep executive-managed pages distinct from kernel-managed pages, the executive allocates pages from a special segment in the executive's own context, the *executive-managed heap*. The executive_brk() and executive_sbrk() calls allow the executive to change the size of this segment in the same way that CMOST's Unix-like brk() and sbrk() work with the standard heap. Allocation on the executive-managed heap is always rounded up to the next multiple of the page size, so that an integral number of pages are allocated. The virtual addresses of these pages in the executive's context are the *primary mappings* (i.e., handles) which the executive uses to refer to these pages across the kernel interface. Only pages in the executive-managed heap—referred to as *executive-managed pages*—may be aliased via add_mapping() or have their memory tag values changed.

The *executive virtual heap* allows the executive to manage a region of its own address space the same way that it manages subcontexts. The executive_vbrk() and executive_vsbrk() calls simply reserve a contiguous collection of virtual pages, but do not allocate physical pages behind them. The executive can then alias these virtual pages to pages in the executive-managed heap (possibly with different attributes, e.g. read-only or non-cacheable) without fear that the kernel will grow the standard heap or stack to conflict. This call is not necessary for subcontexts because the executive automatically has complete control of those.

The add_mapping() call creates a *secondary mapping* from virtual address va in context cd to the page at virtual address pp, i.e. it aliases cd:va and 0:pp, where pp must point to an executive-managed page. The mapping attributes (protection and cacheability) are set according to attr. To prevent interference between the kernel and executive, if cd is zero then va must be in the executive virtual heap. The alias count for the corresponding physical page is incremented.

The change_pg_attr() and delete_mapping() calls allow the executive to change the attributes of and delete mappings, respectively. Only mappings created via add_mapping() can be modified or deleted. delete_mapping() also decrements the physical page's alias count.

3.1.2 Page motion callbacks

The two page-motion callbacks—page_going(), invoked when the kernel must reclaim an executive-managed page, and page_coming(), invoked when a page returns—serve a dual role. First, they allow the kernel and executive to cooperate in physical memory management, similar to the way scheduler activations allow management of physical processors in a shared-memory multiprocessor [2]. By explicitly saving and restoring data in response to page_going() and page_coming() calls, the executive can control exactly which data are resi-

dent in physical memory. Second, the callbacks significantly reduce the kernel's bookkeeping requirements, by giving the executive responsibility for maintaining secondary mappings and memory tags across physical page movements.

The two page motion callbacks only affect executive-managed pages; the executive must register the callbacks before allocating pages on the executive-managed heap. When the kernel decides to reclaim an executive-managed page (e.g., to allocate it to a different process), it notifies the appropriate executive using the page_going() callback. In general, the kernel will call page_going() with the argument pp set to NULL, indicating that the executive can select any executive-managed page for reclamation. The executive can apply its own replacement policy and return the selected page-aligned pp as the return code. By default, the kernel discards the page contents; however, the executive may request that they be saved (i.e. moved to backing store) by overloading the return code (setting the least-significant bit to one). Occasionally, the kernel may need contiguous physical pages—e.g., for the CM-5 vector units—requiring it to reclaim a specific executive-managed page. In this case, the argument pp points to the selected page, and only the least-significant bit of the return value is meaningful.

The executive must use delete_mapping() to delete all secondary mappings for the selected page before returning, at which point the kernel removes the primary mapping from pp to the physical page. Note that virtual address pp is still "in use" as it uniquely identifies this page and will be provided as the argument to a future call to page_coming().

The kernel returns an executive-managed page via the page_coming() callback. The primary mapping is recreated, i.e. the virtual address pp again maps to a physical page, though not necessarily the same physical page as before the page_going() call. If the executive returned a zero in the LSB on the previous call to page_going(), the page has been zeroed; otherwise, the contents have been restored. In either case, the physical memory tags have been cleared. This call allows the executive to restore any secondary mappings and/or memory tags for the page. Secondary mappings could also be restored on demand.

These callbacks allow user-level management of physical memory: even when the kernel reclaims a specific physical page, the executive can choose the data that get replaced at the expense of additional copying and page-table manipulation. Alternatively, the executive can let the kernel save and restore data in evicted pages. The executive can always regain access to a "gone" page by dereferencing pp and causing a fault. The kernel will handle the fault by obtaining a free page (possibly by calling page_going() on this or another executive), restoring the old data (if necessary), and calling page_coming() to signal the return of the needed page.

Because the executive is untrusted, the kernel cannot rely on it to delete all secondary mappings on a page_going() callback. Conversely, it must guarantee that these mappings are removed—otherwise, the executive may retain an alias to a physical page re-allocated to a different process. A brute-force solution is for the kernel to automatically delete all secondary mappings. However, this approach requires that the kernel maintain all of the reverse translations, duplicating state already maintained by the executive. Instead, we require that the executive delete all secondary mappings to the selected page before returning from the page_going() call or face process termination. Process termination guarantees that all mappings are deleted, because the process and all its sub-contexts are destroyed. Thus the kernel only need know *how many* secondary mappings exist, but not their individual identities. A simple counter per physical page, incremented on each add_mapping() call and decremented on each delete_mapping call, is sufficient to maintain this state.

In addition, to protect against deadlock or infinite loops in the executive, the kernel

requires that all callbacks be completed within a fixed time. The kernel sets a virtual timer before invoking a callback; if the timer expires before the callback returns the executive's process is terminated.

3.2 Execution management

Creating and manipulating address spaces is uninteresting without the ability to execute within them. Two calls suffice to provide this functionality. The kernel call jump_to_ctx() causes the current thread of execution to switch into the specified context. When the thread executing in the subordinate context encounters a fault (either an instruction fault or an explicit software trap), control resumes in the executive's context via the ctx_fault() callback.

Because the jump_to_ctx() call continues the current thread in a different context rather than creating a new thread, the contents of the register file are largely unchanged across the switch. The struct regs structure passed as an argument to jump_to_ctx(), though implementation-dependent, conceptually consists of only the program counter and stack pointer in the new context and the original contents of the registers required to pass the three arguments of jump_to_ctx() into the kernel.[2] The third argument, stackp, specifies a stack in the executive context to use when ctx_fault() is invoked. From the executive's perspective, jump_to_ctx() does not return. After a thread passes through a jump_to_ctx() call, it is in a subordinate execution context where all faults are handled by the executive.

When the thread executes a trapping instruction (either due to a fault or an explicit software trap), control resumes in the executive's context at the ctx_fault() entry point. The first argument indicates the type of fault and the second passes back a pointer to the same structure originally passed to jump_to_ctx(). The state saved is exactly the state required by jump_to_ctx(), so supplying this buffer unmodified as the second argument to jump_to_ctx() restarts execution in the other context at the faulting instruction.

Additional arguments are passed from the kernel to the executive depending on the type of the fault—for example, an MMU fault will also provide the virtual address of the access and the nature of the fault (invalid address, protection violation, etc.).

We have, to the greatest extent possible, separated thread management issues from this interface. However, with the addition of thread management code, this simple interface is sufficient for the executive to behave as a multiprogrammed operating system. Implementing a threads package on top of this interface simply requires code to allocate and manage multiple stacks, and to save and restore the registers not contained in the struct regs structure. For example, a context switch is as simple as having the ctx_fault() handler save and restore the CPU registers and call jump_to_ctx() with a different context descriptor and struct regs pointer. Separating the thread and context abstractions gives the executive flexibility, e.g. to support a kernel thread abstraction within its sub-contexts. If the underlying kernel provides a programmable timer interrupt, the interrupt can be made to appear through the ctx_fault() entry as well, making the multithreading preemptive.

The executive does not normally handle its own faults, except for those to the executive virtual heap. Having the kernel handle the executive's faults facilitates demand paging of the executive's text and data, and makes growing the executive's stack the same as for any other user process. If the executive wants to handle its own faults, it can call jump_to_ctx() with the first argument set to zero (its own context). This creates a singular situation where the thread is in a subordinate execution context—so faults still invoke the ctx_fault()

[2] On the SPARC architecture, this structure also contains the NPC (to resume after faults in delayed branch slots) and the condition codes (since these cannot be saved and restored from user mode).

Function	Lines of C	
	PN	CP
executive_{v}{s}brk		
Total, four calls	10	35
create_ctx	134	0
Page table initialization	17	11
Process termination	90	0
Page callback support	222	426
Total	473	472

(a) C language additions

Function	Machine Instructions
add_mapping	187
change_pg_attr	108
delete_mapping	123
jump_to_ctx	50
ctx_fault	40
set_page_motion_cbs	7
set_ctx_fault_cb	5
Total	520

(b) PN assembly additions

Table 2: Code added to CMOST to implement executive interface.

callback—but not a subordinate addressing context. If the executive decides that the kernel should handle a specific fault, it need only retry the faulting instruction from within its fault handler. There is no danger of a recursive call to ctx_fault() because any fault encountered by the executive's handler will be handled directly by the kernel.

Subordinate execution contexts cannot be nested because jump_to_ctx() is a system call implemented as a software trap; "recursive" calls will show up in the executive via ctx_fault(). As with any other traps, system calls made in a subordinate execution context can be forwarded to the kernel simply by re-executing the call in the executive. The only complication occurs with pointer arguments, since the kernel will interpret these in the executive, rather than subordinate, context.

4 Implementation

We have implemented the executive interface in CMOST version 7.2 Beta 1. As shown in Table 2, the entire interface required less than one thousand lines of C and just over 500 machine instructions. Only jump_to_ctx() and ctx_fault() required assembly-level coding; the other functions were implemented in assembly to improve performance. Although the majority of the additional code is in the PN kernel, most of the complexity lies in the CP portion. This follows from the centralized structure of CMOST: the PN kernel implements only mechanisms, while all policy decisions occur on the CP.

The executive interface requires very little additional kernel state. The PN kernel requires a few additions to the process control block (PCB) and an array with an entry per physical page. The PCB maintains the entry points and stack addresses for the callbacks and the struct regs and stackp pointers from the last jump_to_ctx() call (for use in the subsequent ctx_fault() callback). The array, indexed by physical page number, contains the alias count for each page and a pointer field. The pointer field is used to maintain a linked list, whose head is in the PCB, of all physical pages in the executive-managed heap to facilitate resetting the alias counts when the executive process terminates.

We also added two new memory segments to every process: the executive-managed heap and the executive virtual heap. Both segments have special semantics:

- Any time the CP decides to move a page in the executive-managed heap, it must first

Function	Time[3] cycles (μs)	Function	Time[3] cycles (μs)
executive_sbrk		create_ctx	19K (575)
with CP communication		add_mapping	
alloc 1 page	1.6M (48 ms)	no table allocation	359 (11)
alloc 100 pages	4.9M (148 ms)	alloc level 3 table	1166 (35)
w/o CP communication	20K (606)	alloc level 2 & 3 tables	2641 (80)
executive_vsbrk		change_pg_attr	340 (10)
with CP communication	1.4M (42 ms)	delete_mapping	855 (26)
w/o CP communication	20K (606)	jump_to_ctx	180 (5)
set_page_motion_cbs	117 (4)	ctx_fault	154 (5)
set_ctx_fault_cb	108 (3)		

Table 3: Performance of executive interface calls on the CM-5.

invoke the executive's page motion callbacks.

- Allocations in the executive virtual heap segment are not backed by physical memory. This segment simply provides a region in the executive's virtual address space that is guaranteed not to conflict with regions used by the control processor. Also, faults to this segment are always handled by the executive.

On the CP, two fields were added to the per-physical-page structure to record the process ID and virtual address of each physical page that is allocated to an executive-managed heap segment. This information is required to perform the page_going() callback when the physical page needs to be moved.

Table 3 summarizes the performance of the executive interface calls, as measured using the CM-5's cycle counter and averaging tens of iterations.[3] The executive_{v}{s}brk() and create_ctx() calls are implemented as full traps, where the user's register windows are flushed, execution switches to the kernel stack, and interrupts are re-enabled. All other calls are implemented as "fast" traps, and execute without flushing any windows or re-enabling interrupts. The overheads for the two types of traps are approximately 300 μs and 3 μs, respectively.

The callback registration functions simply store their arguments in fields in the process's PCB. No validation is required since illegal values can at worst cause an immediate fault on the invocation of a callback, which will terminate the executive process.

4.1 Memory management

4.1.1 Context management calls

The executive_{v}{s}brk() calls are implemented in the same fashion as CMOST's brk() and sbrk(). The first node requesting a page generates a system-wide interrupt, which causes the control processor to grow the appropriate segment on all nodes. Subsequent requests on other nodes can be satisfied locally by returning pages from the now-larger segment.

[3]This information was derived using a pre-release version of CMOST (7.2 Beta 1 of Feb. 1993). Performance on released versions may be significantly different.

The create_ctx() call allocates a SPARC hardware context number and an empty level-one table (the SPARC has a three-level page table structure). A call to add_mapping() validates its arguments, performs the page table insertion (allocating level two and three tables as necessary), and increments the alias counter for the physical page. Both change_pg_attr() and delete_mapping() validate arguments and do a page table walk, with the latter also decrementing the appropriate alias counter. None of these calls require communication from the node to the control processor.

Because the CM-5 node has a virtually-tagged writeback cache, delete_mapping() must also flush data brought in using the deleted mapping. This requires iteratively flushing 128 cache lines, causing it to take significantly longer than change_pg_attr().

4.1.2 Page motion callbacks

The page motion callbacks account for most of the design complexity. Our current implementation focused on minimizing changes in the CP code. As a result, our implementation is influenced by several existing CMOST features:

- Swapping is supported, but not demand paging. All of a process's pages must be in memory before it is allowed to run.
- Memory management is performed entirely on the control processor, which assumes that the memory maps of all nodes are identical. When the CP reclaims a page, it must select a specific physical page and force all nodes to release it, even if there is no need for physical contiguity. In other words, in this implementation, page_going() is never called with a NULL argument because the CP cannot deal with different nodes freeing different pages.
- Both the PN kernel and the CP portion of CMOST are single-threaded, i.e. there is only one stack in each. In addition, the PN kernel does not maintain any kernel stack state across communications with the CP.

While this implementation satisfies our needs for WWT, and serves as a proof-of-concept for the executive interface, the CP code requires more radical changes for a clean and efficient implementation.

In CMOST, pages are freed for two reasons: i) to satisfy an allocation request for a currently executing process or, ii) to swap in an idle process. In either case, the CP performs the necessary memory management operations while the PNs are idle, waiting for instructions to resume. Page moves not requiring callbacks are performed immediately, but those requiring callbacks are recorded and deferred. Before resuming the user process, the CP performs the deferred callbacks, scheduling the affected processes (executives) as needed.

The callbacks are executed in the executive context by invoking the registered callback function using the registered stack pointer. The callback cannot be run on the current process stack because it could be in a different address space (i.e. if the process was suspended while running in a subcontext). A fault in the callback will result in the executive's termination. When the callback returns, the PN kernel either executes another callback pending for this executive, or waits for the other nodes to complete. Once all callbacks are done for this executive (across all nodes), the scheduler is re-invoked to run the next executive with pending callbacks. When all deferred callbacks have executed, the originally scheduled process can run.

Because the executive is scheduled for callbacks the same way it is scheduled for normal execution, there are only a few constraints on callback execution. First, the callback cannot allocate memory of any kind since this could create a circular dependency. Second, each callback is provided at most one scheduling quantum; the executive is terminated if the quantum expires during a callback. Third, the callback also cannot call any blocking kernel function, since this would interfere with the callback timeout mechanism. Finally, the current implementation does not allow the callback to use the CM-5 network interface. This last restriction is not inherent to the architecture, but would add significant complexity and overhead for a feature our application would not use.

Our implementation is designed to support swapping, but we have not yet tested this portion of the code. However, the CM-5 vector unit architecture severely constrains physical memory allocation, causing CMOST to frequently reallocate specific pages, moving data from one physical page to another. By treating these page moves as a swap out followed immediately by a swap in, we have completely exercised the callback mechanisms.

4.2 Execution management

The subordinate execution context is simply the same CMOST process executing with a different trap vector and (perhaps) a different hardware MMU context. All trap vector entries, except hardware interrupts, jump to the ctx_fault() kernel stub.

The jump_to_ctx() and ctx_fault() functions are implemented as "fast" traps, i.e. they execute without re-enabling interrupts in a partial SPARC register window. In addition to loading state from the struct regs structure, (or storing it in the case of ctx_fault()), both calls must change the hardware context, manipulate the register window mask, and change the trap vector base address. jump_to_ctx() requires ten extra instructions because it must read the processor status register, mask in the desired condition codes, and write it back.

4.3 Other implementation issues

The executive needs to specify a stack for all interrupt or signal handlers, as it does for the page motion callbacks, since the current process stack may not exist in the executive's own address space. We have not added this extension yet, but it would be simple to do so.

The cache controller used on the CM-5 node (Cypress 604) is a 64KB direct-mapped virtually-tagged cache. The hardware will handle aliases that map to the same cache block, but cannot guarantee consistency otherwise. To avoid cache flushing, aliases must be congruent modulo 64K. Rather than complicating the interface with this issue, we force the executive to deal with it on its own. We have added an additional call to the PN kernel, int cache_flush_page(int cd, void *pp), which flushes the specified page from the SPARC cache. This allows the executive to make the tradeoff between keeping aliases congruent and performing cache flushes according to its own needs.

5 Discussion

While the executive interface is interesting in isolation, it is more striking when considered in the context of the CM-5 and CMOST. The resulting kernel structure is—we believe—unique. In most microkernels designed for parallel systems, nodes are fundamentally autonomous. Cooperation, e.g. for gang-scheduling, occurs as a policy at a higher level

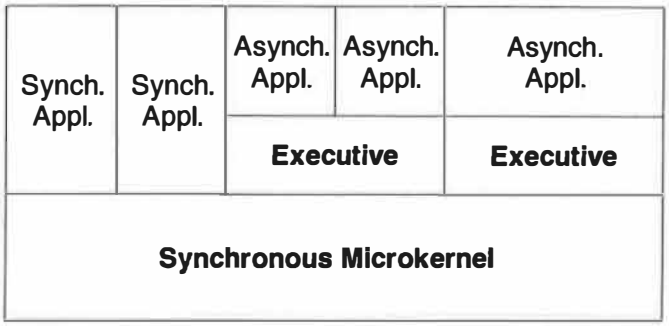


Figure 1: Mixed-model parallelism using the executive interface.

of abstraction. Our extended version of CMOST turns this structure on its head: the control processor maintains central, synchronous control of physical memory and scheduling. The control processor still forces all processors to context switch simultaneously; however, some of the processes may now be executives. Executives, because of the autonomy provided by the executive interface, can schedule execution in their subcontexts however they choose. This flexibility can be used to support applications or groups of applications which may benefit from more dynamic allocation and scheduling policies (see Figure 1). Thus our extended CMOST provides autonomy on top of synchrony, rather than the more traditional alternative of synchrony on top of autonomy.

The CMOST/executive structure was motivated by our implementation of the Wisconsin Wind Tunnel on the CM-5. However, the resulting structure is arguably the right way to structure an operating system for large-scale parallel machines. Efficiently supporting fine-grain parallel applications requires a global perspective for resource allocation, because a page fault or scheduling delay on one node can seriously impact the performance of the entire application. Centralizing control, as in CMOST, makes global resource allocation significantly easier. For example, CMOST's guarantee that one user process runs simultaneously on all nodes allows direct user access to the CM-5's network interface hardware, avoiding costly system calls for message operations.

Operating systems that fail to efficiently manage global resources will have a particularly difficult time exploiting hardware features such as the CM-5's control network [9], which performs a global barrier or reduction in a few microseconds. Because hardware barriers are both cheap and fast—they are essentially AND-gates—we expect them to appear in most future parallel machines.[4] The operating systems for these machines must be able to exploit this hardware to efficiently execute fine-grain data-parallel codes. We believe this may prove easier with a synchronous microkernel structure, rather than a more traditional asynchronous kernel structure.

While the CMOST/executive structure supports timesharing among different execution models, a hierarchical control structure can integrate space-sharing as well. For example, a central scheduler on a 128-node machine can reserve some time-slices for 128-node synchronous applications and some for 128-node applications or sets of applications with more dynamic executive-managed scheduling. The remaining time-slices can be delegated to two other schedulers, each of which can recursively do identical centralized scheduling within

[4]The Cray T3D also has hardware support for global barriers; barriers are actually faster than remote memory operations on this machine [14].

disjoint 64-node processor groups.

In order for a single executive to manage multiple users' applications, the executive must be run with some additional privilege, e.g. as a Unix "setuid root" process, to access system resources with the effective permissions of the user on whose behalf the current application is being executed. Such an executive would also need a way to adjust its scheduling priority within the kernel so that processing resources can be fairly allocated across all user jobs, whether they are executing directly under the kernel or are one of several running under an executive.

6 Related Work

The interface described in this paper was motivated by the needs of the Wisconsin Wind Tunnel. The centralized structure of CMOST, with all policy enacted on the control processor, was insufficient to support the user-level fine-grain distributed shared memory needed by WWT. The executive interface extends CMOST to provide nodes with limited autonomy in the way they manage their virtual address spaces and physical memory. This interface is interesting from two different perspectives: on its own, as a means of exporting memory-management functions to the user of a uniprocessor; and in conjunction with CMOST, as a means of supporting multiple models of application parallelism on a single machine.

6.1 Uniprocessor aspects

The executive interface provides a complete set of low-level virtual memory functions. The interface is simpler and lower-level than the virtual memory interfaces of either Mach [12] or Chorus [1], which both impose significant semantics on the use of memory. To the first order, the executive interface merely exposes the underlying hardware mechanisms to the user in a protected manner.

The executive interface is similar to the "inferior spheres of protection", described by Dennis and Van Horn [6]. Their execution model allowed processes to create subcontexts, initiate execution within them, and handle any resulting faults. The primary difference is our page orientation rather than their more general segments and capabilities.

More recently, Probert, et al, proposed SPACE, an object-oriented operating system [11]. SPACE allows applications to create, manipulate, and execute within *spaces*, i.e., address spaces, thereby facilitating protected objects. However, SPACE is much more general than our interface, allowing different "executives" to manage different parts of a single address space.

Appel and Li surveyed the most common uses of user-level virtual memory, and identified the set of primitives needed by these applications [3]. The set includes primitives to modify protection on pages and create aliases within an address space; however, they did not include the ability to create new address spaces, nor get callbacks when pages are reclaimed by the kernel.

Using page motion callbacks to manage physical memory allocation is analogous to using scheduler activations to manage physical processor allocation [2]. Both provide the user with notification of kernel allocation decisions so that the application can adapt knowledgeably to its new circumstance. A key difference is that the page_going() callback notifies the user *before* the page is taken away, while the scheduler activation model notifies the user *after* a processor has been taken away. This adds some complexity to the page motion callbacks (i.e. the necessity for the kernel to enforce a finite completion time), but reflects

a fundamental difference between memory and processors: it is reasonable to have the user allocate space to save a processor's state in case the kernel takes it away, but nonsensical to apply the same principle to memory pages.

The Mach external memory manager interface is similar to our page motion callbacks. However, if an external memory manager does not remove a page in a timely fashion, the Mach kernel can always write the page to backing store using the default memory manager. Our interface does not permit this, because the kernel cannot clean up secondary mappings itself nor can it permit them to point into another process.[5]

6.2 Multiprocessor aspects

User control over multiprocessor scheduling with the intent of supporting multiple models of parallelism (including gang-scheduling) is provided in Mach by a processor allocation server [4]. In this model, an application requests a certain number of processors to create a "processor set" to which threads can be bound. The binding of actual processors to processor sets is performed by a privileged user-level server. The server can be modified to support site- or usage-specific policies, but there can only be one per platform. In our scheme, a process requiring a fixed number of processors can be handled simply by scheduling it at the appropriate level in the hierarchy. A process with more dynamic needs could be served by an appropriate executive that balances its requirements by scheduling it in conjunction with other applications having similar dynamic parallelism. The fundamental differences are that our model provides gang-scheduling more as the rule than the exception, and allows for multiple executives running simultaneously to support multiple abstractions.

The hierarchical integration of time- and space-sharing discussed in Section 5 is similar to Feitelson and Rudolph's distributed hierarchical control [7], except that they assume a hardware hierarchy of control processors, while we believe a software hierarchy of control processes may be just as effective. Also, their model does not have the executive interface to provide different scheduling models underneath the global hierarchical structure.

7 Conclusion

The executive interface exports a complete, abstract model of virtual memory management to a user process, including the ability to create, manipulate, and execute in multiple address spaces. The interface allows the user process to participate in physical memory management using page motion callbacks. The callbacks also serve to minimize the kernel complexity of implementing the interface.

Giving a user-level executive the ability to define a complete virtual memory environment in a protected fashion allows multiple executives providing multiple process abstractions to coexist on a single system. Though interesting from a uniprocessor perspective, it is more significant in the context of large-scale multiprocessors. Instead of having the operating system view the machine as a set of autonomous nodes upon which coordination mechanisms must be imposed, it can start with a global perspective and selectively delegate nodes in both space and time to executives which allow increasing amounts of autonomy. The resulting structure combines the advantages of centralization and decentralization: the underlying global perspective simplifies efficient support of fine-grained synchronous (e.g.

[5] Our kernel could write the page to backing store, so long as it also saved and restored the memory tags and guaranteed to return it to the *original* physical page before resuming the process.

data-parallel) applications and management of global resources such as barrier hardware, while executives provide the flexible support for other application models that a completely centralized system lacks.

Acknowledgements

Mark Hill, Frans Kaashoek, Jim Larus, Bart Miller, Yannis Schoinas, and Marv Solomon provided helpful comments that greatly improved this paper.

References

[1] Vadim Abrossimov and Marc Rossier. Generic Virtual Memory Management for Operating System Kernels. In *Proceedings of the Twelveth ACM Symposium on Operating System Principles (SOSP)*, pages 123–136, December 1989.

[2] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.

[3] Andrew W. Appel and Kai Li. Virtual Memory Primitives for User Programs. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 96–107, April 1991.

[4] David L. Black. Scheduling Support for Concurrency and Parallelism in the Mach Operating System. *IEEE Computer*, 23(5):35–43, May 1990.

[5] M. D. Canon, D. H. Fritz, J. H. Howard, T. D. Howell, M. F. Mitoma, and J. Rodriguez-Rosell. A Virtual Machine Emulator for Performance Evaluation. *Communications of the ACM*, 23(2):71–80, February 1980.

[6] Jack B. Dennis and Earl C. Van Horn. Programming Semantics for Multiprogrammed Computations. In *ACM Programming Languages and Pragmatics Conference*, August 1965.

[7] Dror G. Feitelson and Larry Rudolph. Distributed Hierarchical Control for Parallel Processing. *IEEE Computer*, 23(5):65–77, May 1990.

[8] Robert P. Goldberg. Survey of Virtual Machine Research. *IEEE Computer*, 7(6):34–45, June 1974.

[9] Charles E. Leiserson, Zahi S. Abuhamdeh, David C. Douglas, Carl R. Feynman, Mahesh N. Ganmukhi, Jeffrey V. Hill, W. Daniel Hillis, Bradley C. Kuszmaul, Margaret A. St. Pierre, David S. Wells, Monica C. Wong, Shaw-Wen Yang, and Robert Zak. The Network Architecture of the Connection Machine CM-5. In *Proceedings of the Fifth ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, July 1992.

[10] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[11] D. Probert, J. Bruno, and M. Karaorman. SPACE: A New Approach to Operating System Abstraction. In *Proceedings of the International Workshop on Object Orientation in Operating Systems*, October 1991.

[12] Richard Rashid, Avadis Tevanian, Michael Young, David Golub, Robert Baron, David Black, William J. Bolosky, and Jonathan Chew. Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. *IEEE Transactions on Computers*, 37(8):896–908, August 1988.

[13] Steven K. Reinhardt, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, and David A. Wood. The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers. In *Proceedings of the 1993 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 48–60, May 1993.

[14] S. L. Scott. Personal communication, June 1993.

[15] Burton J. Smith. Architecture and Applications of the HEP Multiprocessor Computer System. In *Proc. of the Int. Soc. for Opt. Engr.*, pages 241–248, 1982.

[16] Thinking Machines Corporation. *Connection Machine CM-5 Technical Summary*, October 1991.

RT-IPC: An IPC Extension for Real-Time Mach

Takuro Kitayama[1], *Tatsuo Nakajima*[2], *and Hideyuki Tokuda*[1]

[1] *School of Computer Science, Carnegie Mellon University*
{takuro, hxt}@cs.cmu.edu
[2] *Japan Advanced Institute of Science and Technology*
tatsuo@jaist-east.ac.jp

Abstract

Interprocess communication (IPC) provides the fundamental mechanism for the Mach microkernel to be extensible and flexible. Mach IPC provides efficient communication mechanisms for many applications. However, it does not provide sufficient functionality for real-time applications which have rigid timing constraints among threads. In Real-Time Mach (RT-Mach), we have extended Mach IPC to be *priority inversion free* for real-time applications.

This paper describes the Real-Time IPC (RT-IPC) facilities, its implementation, and the evaluation results. We used the Distributed Hartstone (DHS) real-time benchmark for the evaluation and the results show that RT-IPC can reduce priority inversion and improve CPU utilization for real-time applications.

1. Introduction

In traditional operating systems, it is difficult to support real-time applications such as continuous media applications and mobile computing, due to the lack of facilities which can manage time constrained resources, such as processor cycles, memory, communication ports, shared variables, and so on. To support real-time applications, we need time-driven resource management for all such resources.

Real-Time Mach (RT-Mach) has been developed at CMU to provide a common distributed real-time computing environment[19]. In RT-Mach, we have developed a real-time thread model where a periodic activity can be easily defined by explicitly specifying a timing attribute. It also provides real-time scheduling services, real-time synchronization[18], and high resolution clocks and timers[14].

The original Mach microkernel provides a port-based IPC mechanism. However, it is designed and optimized to provide fair services to all type of application programs[2, 3]. In the timesharing environment, it is very rare that two or more messages are queued in one message queue, usually, messages are delivered from a sender to a receiver without queueing, i.e., the average queue length would be less than one.

However, in the real-time environment, several messages can be queued in one queue and wait to be serviced by the receiver. For example, if a video playing server is not fast enough to process several requests within a certain time, some requests must be queued. Furthermore, those messages

This research was supported in part by the U.S. NRaD under contract number N66001-87-C-0155, by the Office of Naval Research under contract number N00014-84-K-0734, by the Defense Advanced Research Projects Agency, ARPA Order No. 7330 under contract number MDA72-90-C-0035, and by the Federal Systems Division of IBM Corporation under University Agreement YA-278067. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of NRaD, ONR, DARPA, IBM, Northrop, Bellcore, or the U.S. Government.

should have a priority according to user's interest. Data which will be displayed on a background window can be delayed, but data which is currently in an active window should be displayed within a certain time. We need a mechanism which can guarantee message delivery in a timely fashion.

For real-time synchronization, there has been developed many mechanisms to avoid unbounded priority inversion problems[12, 15]. However, the most of the real-time synchronizer are designed without integrated with real-time IPC. In general, the priority inversion problem in client-server communication is more serious one, since the length of priority inversion tends to be much longer than that of synchronization. In this paper, we propose a Real-Time IPC (RT-IPC) mechanism to support real-time applications, describe its implementation in RT-Mach, and show the performance evaluation results of RT-IPC. In Section 2, we show the overview of RT-Mach and the original Mach IPC, then describe the problems with the conventional IPC mechanism. Section 3 discusses our RT-IPC model. Section 4 shows the implementation of RT-IPC in RT-Mach. Section 5 shows the performance evaluation of RT-IPC. Related work is discussed in Section 6, and we conclude in Section 7.

2. Motivation

In this section, we summarize Real-Time Mach, and the original Mach IPC, then we describe the problem with the conventional IPC mechanism.

2.1. Overview of Real-Time Mach

The objective of Real-Time Mach is to develop a real-time version of Mach which supports a predictable real-time computing environment. The major features of Real-Time Mach are summarized as follows:

Real-Time Thread Model : A thread is defined for a real-time or non-real-time activity. For a real-time thread, additional timing attributes must be defined by a *thread attribute*. A real-time thread can be also defined as a *periodic* or *aperiodic* thread based on the nature of its activity.

Real-Time Scheduler : In real-time operating systems, thread scheduling plays an important role in managing the system resources in a timely fashion. The RT-Mach scheduler provides us a good flexibility by supporting multiple scheduling policy. Currently, The RT-Mach scheduler supports Time-Sharing, Fixed Priority, Rate Monotonic, Deadline Monotonic, and Earliest Deadline First. A scheduling policy can be set as one of *processor set attributes*.

High Resolution Clock and Timer : Clocks are devices which measure the passage of time and support the use of timers to a particular degree of accuracy. A clock device can be used for accurate timestamps, measurements of CPU usage, or for basing representations of the time of day. Timers are active objects which allow users to synchronize with time in a variety of ways. Timers are kernel-exported objects with three properties; an expiration time, a synchronization action to be taken, and an activity state.

Real-Time Synchronization : Traditional synchronization primitives use FIFO ordering among threads waiting to enter a critical section. In real-time environment, however, FIFO ordering often creates a priority inversion problem where a low priority thread blocks a high priority thread. RT-Mach supports various synchronization policies to avoid unbounded priority inversion.

2.2. Overview of Mach IPC

IPC plays a very important role in the Mach microkernel to provide communication between tasks, especially clients and servers. Since the Mach microkernel provides limited services of its own, a task needs to communicate with many other tasks which provide a variety of services such as file systems

and network services. These communication channels are called *ports*. A port is a unidirectional channel consisting of queue that holds *messages*. A message is a typed collection of data. A port is named by *port rights* held by tasks. A task has a *port name space* which collects port names. A task can manipulate a port only if it holds appropriate port rights. Only one task can hold the *receive right* for a port. This one task is allowed to receive message from the port. Multiple tasks can hold *send right* to the port that allow them to send messages into the queue. A task communicates with another task by building a data structure that contains a set of typed data elements[1], and then performing a message-send operation on a port for which it holds send rights. The message is queued onto the message queue in FIFO order. At some later time, the task with receive rights to that port will perform a message-receive operation. The message is logically copied into the receiving task. Multiple threads within the receiving task can receive messages from a given port, but only one thread will receive any given message.

2.3. Basic Issues

IPC is also heavily used in the RT-Mach environment to provide a variety of services by server programs. The original Mach IPC facilities provide fair services for timesharing applications. However, in the real-time environment, this causes unbounded priority inversion problems. In order to build servers for providing various real-time services on top of RT-Mach, we must have a real-time version of IPC.

There are four categories where problems occur with the original Mach IPC: message queueing order, priority hand-off, priority inheritance, and message distribution. In this section, we discuss the four issues.

2.3.1. Message Queueing Order

For timesharing systems, a FIFO queuing policy is acceptable because it ensures fairness. In the real-time environment, however, this may cause the unbounded priority inversion problem where a higher priority message is sent when lower priority messages are on the queue. The processing of the higher priority message has to wait until all processing of lower priority messages are completed. Thus, we need priority-based queueing for real-time applications.

2.3.2. Priority Hand-off

Since the sender and the receiver threads are independent scheduling entities, there should be a mechanism to maintain a priority of receiver when the receiver receives a message. If the priority of the server is lower than that of the sender or message, the processing for the message is interrupted by any other medium priority thread. If the priority of the server is higher than that of the sender, then a low priority request may block any other high priority activities. The receiver thread must propagate the priority from the sender or give a specified priority.

2.3.3. Priority Inheritance

The other issue arises when a high priority message is sent while the server is processing for a lower priority message. The high priority message must be queued on the queue and wait for the completion of the lower priority message processing. At this time, if a medium priority thread becomes runnable, then the receiver thread is preempted by the medium priority thread, even though the higher priority thread is waiting for service. The receiver thread has to inherit the priority from the higher priority sender when the higher priority thread sends a message. This solution is called a priority inheritance protocol[16] and has been developed and applied for real-time synchronization. The basic idea is that a low priority thread inherits the priority of a high priority thread when a high priority thread is delayed by the low priority thread. We need the same mechanism for the real-time IPC. When a high priority thread sends a message, then the priority of the sender should be inherited to the receiver thread.

[1] The user is not required to generate there. MIG (Mach Interface Generator) generates the necessary stub routines.

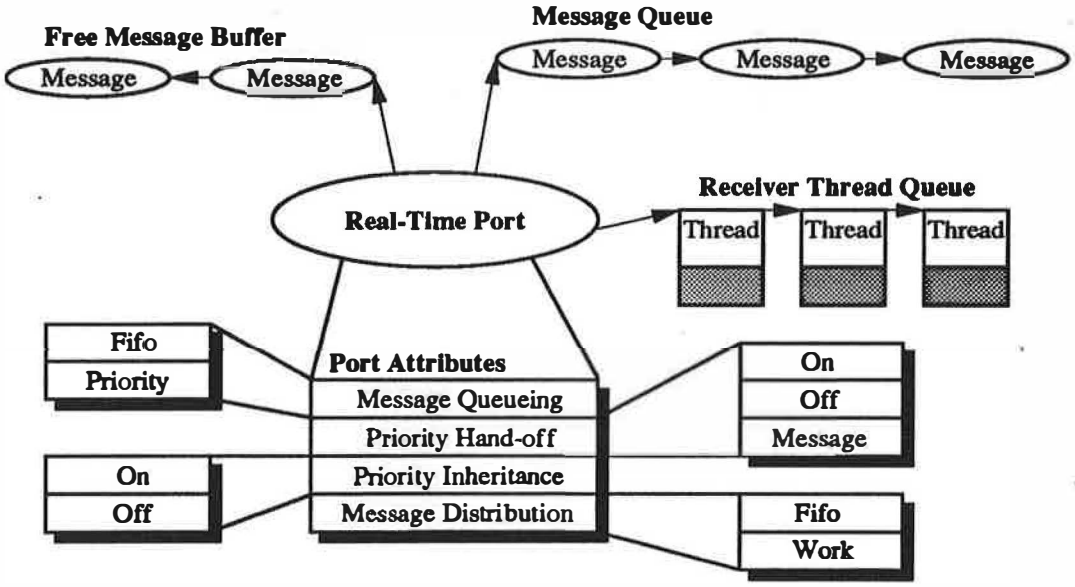


Figure 1: Real-Time IPC Model

2.3.4. Message Distribution

In case there are two or more receivers in a server, there should be a mechanism to determine which receiver should process the message. If there is a sufficient number of receivers running, we do not need to worry about the situation. However, if there are insufficient, we need the mechanisms to select which receiver thread will process the message and bump up its priority if the priority inheritance protocol is activated.

3. Real-Time IPC Model

In order to support real-time applications, we propose the following RT-IPC model. Mach IPC guarantees that messages are received in the order in which they were sent, but RT-IPC could not guarantee that order, since it may cause priority inversions. Our goal is to provide a predictable IPC mechanism for real-time applications.

Figure 1 depicts the basic model of our RT-IPC. We discuss the model focusing on the difference from the original Mach IPC model.

3.1. Port and Receiver Association

RT-IPC supports two kinds of transmission modes. One is synchronous transmission in which the sender is blocked by the receiver until it receives the reply message. The other is asynchronous transmission in which the sender is never blocked by the receiver process, i.e., sender does not wait for the reply message from the receiver. In the both modes, a receiver thread must declare itself to act as the receiver of a port before initiating a communication. By this declaration, the server thread is associated with the port, and its priority will be maintained according to the selected port attribute which is described below.

3.2. Message Buffer

In RT-IPC, message buffers should be allocated before communication to eliminate unpredictable buffer allocation delay. Users need to determine the actual size and number of buffers before initiating

the communication.

3.3. Port Attributes

Related to the queueing and priority manipulation, we define four attributes for the real-time port and provide various policies for each attributes. These attributes are summarized as follows.

Message Queueing : It maintains the message queue ordering. FIFO and priority-based ordering policies can be used.

Priority Hand-off : Priority Hand-off manipulates the receiver's priority when a message is transferred. When it is disabled, the priority of the receiver is not changed. When it is enabled, the priority of the receiver is propagated from that of the sender or given priority according to the selected policy.

Priority Inheritance : It executes the priority inheritance protocol[16]. If it is activated, the server inherits the priority of the sender thread which sent the highest priority message.

Message Distribution : It selects a proper receiver thread when two or more receivers are running. Arbitrary or priority-based(WORK) selection can be specified as a policy. When the arbitrary policy is specified, a receiver thread is chosen in FIFO order. The priority-based policy selects a receiver thread according to a given priority.

3.4. Integration with Synchronization

Many mechanisms have been developed for real-time synchronization to avoid unbounded priority inversions. However, synchronization mechanism alone does not avoid the problems, especially in microkernel environments. Application program could be decomposed into several tasks, and they need to interact and communicate each other. A message may be sent in a critical region, a receiver thread has a critical region, or a sender thread can be also a receiver thread of another message.

In order to avoid unbounded priority inversion between synchronization and IPC, we need to maintain priorities consistently among threads and among all critical region and IPC calls.

For example, there are three threads T_H, T_M, and T_L, where T_H has the highest priority, T_M is medium, and T_L has the lowest priority, respectively. When T_M is in a critical section, T_H is trying to get into the critical section, and T_M sends a message to T_L, then priority hand-off happens in different way between the two transmission modes. In synchronous transmission mode, priority of T_H is propagated to T_L, since T_H has to wait the completion of T_L to go into the critical section. In asynchronous transmission mode, the native priority of T_M is passed to T_L, because T_H will never be block by the computation of T_L.

A similar situation may happen if IPCs are called in a nested fashion. If priority inheritance occurs, the priority of all message receiver threads in the nested call will be affected in synchronous mode. In asynchronous mode, on the other hand, priority inheritance affects only the first receiver thread.

4. Implementation

We have implemented RT-IPC in the RT-Mach microkernel to support the queueing and priority management, which were described in the previous section, to provide a better communication mechanism for real-time applications. In this section, we describe the implementation and interface of RT-IPC.

4.1. Implementation Strategy

The interface must follow the original Mach IPC and the rest of the Real-Time Mach Interface such as Real-Time thread and synchronization to provide the uniform interface for programmers.

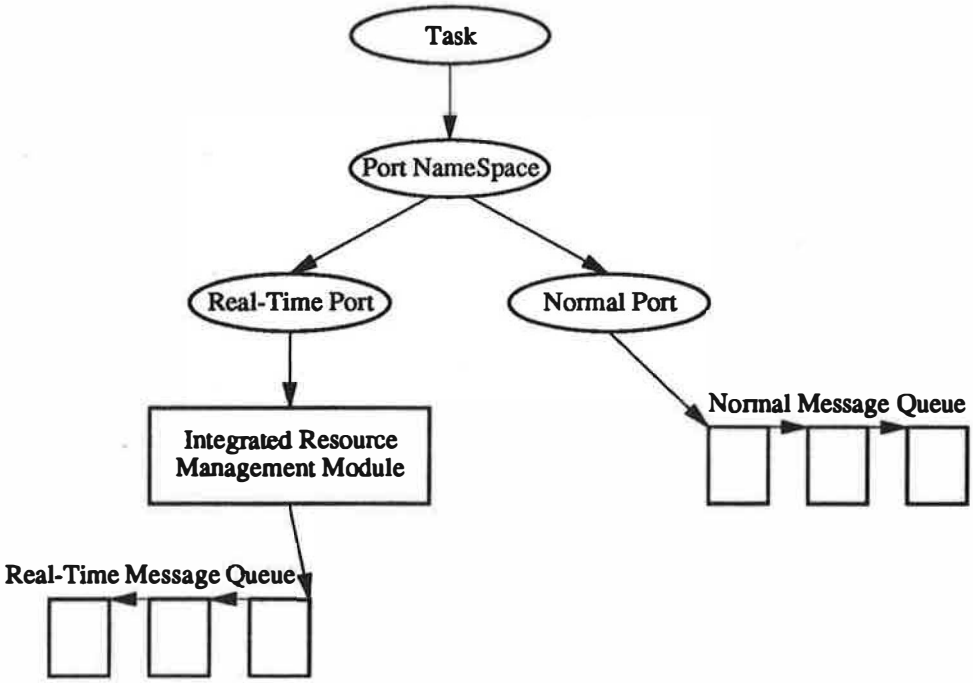


Figure 2: IPC and Real-Time IPC Structure

Figure 2 illustrates the coexistence of the normal IPC and RT-IPC. Each task has one port name space which is sheared by normal IPC and RT-IPC. However, the structure of normal ports and rt-ports are different. A *real-time port(rt-port)* is the real-time version of communication channel(port). A normal port has a message queue to hold waiting messages. A RT-port has some function pointers to the Integrated Resource Management module, which is describe below, instead of having a message queue directly. Each thread can send and receive both normal and real-time messages from and to a normal port or RT-port, if the thread has appropriate right for the ports. Normal ports can handle only normal messages, and RT-ports can handle only real-time messages.

Another main difference is kernel message buffer management. In the original Mach IPC, the microkernel may have one kernel buffer allocated for small size of messages. In RT-IPC, however, all kernel messages are allocated at the port allocation time.

We also consider the effect on the original Mach IPC, it must be minimal. The interface of original Mach IPC has not been changed, we just added some code in the microkernel to check whether the port is a real-time or normal port.

4.2. Basic Structure of RT-IPC

The basic structure of the RT-IPC module in the microkernel is shown in Figure 3. It is roughly divided into two parts. One is the real-time port management module, the other is the Integrated Resource Management (IRM) module. The real-time port management module manages ports, port rights, port name spaces, port sets, and messages in the same fashion as the original Mach IPC[6, 2].

In order to maintain the uniform priority management between RT-IPC and RT-synchronization, we use a common mechanism for message queueing and priority control. The Integrated Resource Management module has been developed for this purpose[10].

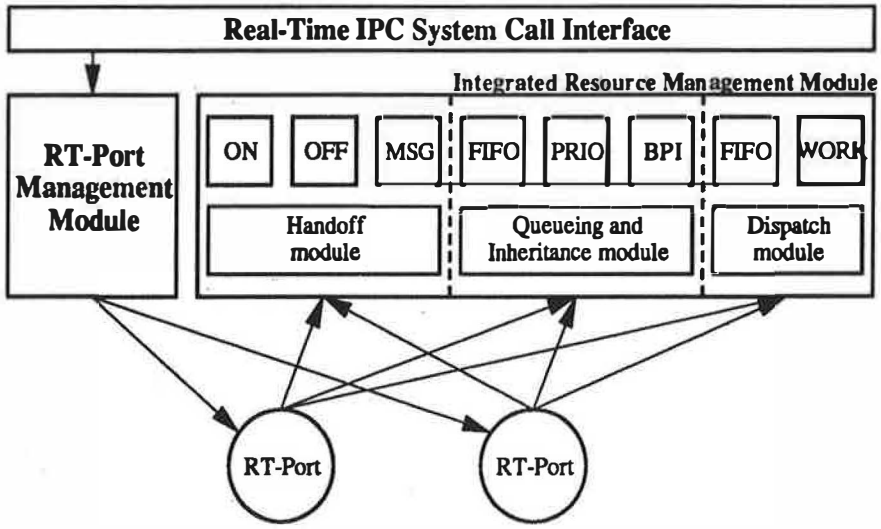


Figure 3: Real-Time IPC Structure

4.3. Integrated Resource Management Module

The basic abstraction in our model is a *resource* and a *user*. A resource represents logical resources such as a critical region or threads in a server. A user represents a schedulable entity which accesses a resource. A resource is manipulated by the two functions: *acquire* and *release*. A user must call *acquire* before using it and *release* should be called after the use. When a user tries to acquire the resource which has been acquired by other user already, the execution of the user is suspended until the resource becomes free. The suspension of the user may cause unbounded priority inversion. In our model, we assume that each user is assigned a priority using the fixed priority scheduling policy such as rate-monotonic scheduling.

The IRM Module is divided into three modules: Priority Handoff Module, Priority Queueing and Inheritance Module, and Dispatch Module. The current version of IRM Module supports the following policy modules.

Priority Handoff Module : This module supports *HANDOFF_OFF*, *HANDOFF_ON*, and *HANDOFF_MSG*. The *HANDOFF_OFF* does not change the receiver's priority. The *HANDOFF_ON* propagates the priority of sender to the receiver. The *HANDOFF_MSG* propagates the priority specified in the message header to the receiver.

Priority Queueing and Inheritance Module : This module supports *PRI_FIFO*, *PRI_PRI*, and *PRI_BPI*. The *PRI_FIFO* maintains the message queue in FIFO ordering. The *PRI_PRI* makes the message queue in priority-based order. The *PRI_BPI* makes priority ordering queue for the waiting messages, and it executes the priority inheritance protocol.

Dispatch Module : This module supports *DISP_FIFO* and *DISP_WORK*. The *DISP_FIFO* module dispatches a free resource if it is available. If not, arbitrary resource is chosen. The *DISP_WORK* selects the receiver by comparing the priority of the receivers and message.

4.4. Extended Interface

In RT-IPC *mach_port_allocate*, *mach_port_allocate_name*, and *mach_msg* cannot be used for real-time ports and real-time messages. Instead of these functions, we created *rt_mach_port_allocate*, *rt_mach_port_allocate_name*, and *rt_mach_msg* to handle real-time ports and real-time messages. Other extension is *rt_mach_port_associate* and *rt_mach_port_not_associate*. These functions are summarized as follows.

```
rt_mach_port_allocate(space, right, rt_port_attr, namep)
rt_mach_port_allocate_name(space, right, rt_port_attr, name)
rt_mach_port_associate(thread, name, priop)
rt_mach_port_not_associate(thread)
rt_mach_msg(rt_msg, option, send_size, rcv_size, rcv_name, time_out, notify)

typedef struct {
    mach_msg_size_t     size;           /* size of buffer */
    unsigned int        nbufs;          /* Number of buffers */
    struct rt_ipc_policy {
        unsigned int    dispatch;       /* Message distribution policy */
        unsigned int    prio_inherit;   /* Priority assignment policy */
        unsigned int    prio_handoff;   /* Priority hand-off policy */
    } policy;
} rt_mach_port_attr;

typedef struct {
    mach_msg_bits_t     msgh_bits;
    mach_msg_size_t     msgh_size;
    mach_port_t         msgh_remote_port;
    mach_port_t         msgh_local_port;
    mach_port_seqno_t   msgh_seqno;
    mach_msg_id_t       msgh_id;
    rt_priority_data_t  msgh_priority;  /* message priority */
} rt_mach_msg_header_t;
```

When a real-time port is allocated, *rt_mach_port_attr* specifies port policies (i.e., handoff, queueing, and dispatch policy), number of kernel message buffers and its size. *rt_mach_msg_header_t* has the same field of the original Mach message header. When HANDOFF_MSG is chosen as the Handoff policy, *msgh_priority* is added to the real-time message header to specifies a message priority.

rt_mach_port_allocate and *rt_mach_port_allocate_name* allocate a real-time port or a port set. The difference between two is whether the port name is given by the system or user.

Before receiving messages, each receiver thread must be associated with a real-time port or port set by calling *rt_mach_port_associate*, and then the thread is recognized as a server thread. In the current implementation, threads can be associated with one port or port set. When DISP_WORK is selected as Dispatch policy, *priop* in the arguments specifies the priority which is used the receiver priority. If a thread tries to associate with a normal port or port set, then *rt_mach_port_associate* returns with KERN_INVALID_NAME.

rt_mach_port_not_associate breaks an association between a receiver thread and the real-time port or port set. After it calls this primitive, the thread cannot receive from the real-time port or port set unless calling *rt_mach_port_associate* again.

rt_mach_msg sends and/or receives a real-time message. The arguments of this function are the same as *mach_msg* which is the original Mach IPC send and receive primitive, except the option argument and the header of the message. If the message is sent in synchronous transmission mode, the MACH_SEND_SYNC flag must be specified in the option argument. If a user tries to send a message to a normal port using *rt_mach_msg*, then the user get an error of MACH_SEND_INVALID_DEST. The Priority inheritance and priority hand-off are in affect until the receiver issues a next message receive call.

Other port manipulation primitives, such as *mach_port_insert_right*, *mach_port_deallocate*, and *mach_port_destroy* can also be used for the real-time port.

5. Performance Evaluation

We measured the transmission cost with different policies to evaluate the low-level RT-IPC primitive cost. We also executed a Distributed Hartstone benchmark program for high-level performance evaluation to compare the effectiveness of various policies.

In the following evaluation, we compared the cases where *Mach IPC*, RT-IPC with *FIFO/OFF*, *PRIO/ON*, and *BPI/ON* policies are used. *Mach IPC* indicates the original Mach IPC, *FIFO/OFF* means measured with FIFO queueing without priority handoff. *PRIO/ON* represents priority-based message queueing with priority handoff. *BPI/ON* means basic priority inheritance with priority-based queueing and priority handoff enabled.

We used a Gateway 2000 4DX2/66E which has a 66MHz Intel 80486DX2 processor and 16 megabytes of memory. The system was running RT-Mach version MK78 with CMU UNIX server version UX39 in a single user mode. Additionally, the network was disconnected and the benchmarks were created with the highest thread priority. We used an Alpha Logic's STAT! timer board to take measurements accurate to 250 nanoseconds.

5.1. Message Transmission Cost

The basic transmission cost of RT-IPC in Figure 4 and 5. The measurement was repeated 1000 times and the averages are taken. Figure 4 shows the round trip transmission cost of RT-IPC and the original Mach IPC where a receiver is ready and waiting for a message. The costs of sending a message while the receiver is servicing another request are shown in Figure 5.

The cost of RT-IPC is 10 to 20% higher than that of Mach-IPC, because of the additional functionality. The differences among the various policies of RT-IPC are due to the different characteristics of the message queueing and priority control policies. The gap between RT-IPC and Mach-IPC are different where the message size is less than 512 bytes and over 512 bytes. This comes from the different IPC path in the microkernel and cache effects. When a message is large, Mach-IPC calls some routines to allocate and deallocate a kernel message buffer, but the RT-IPC does not need this, since the message buffers are allocated previously.

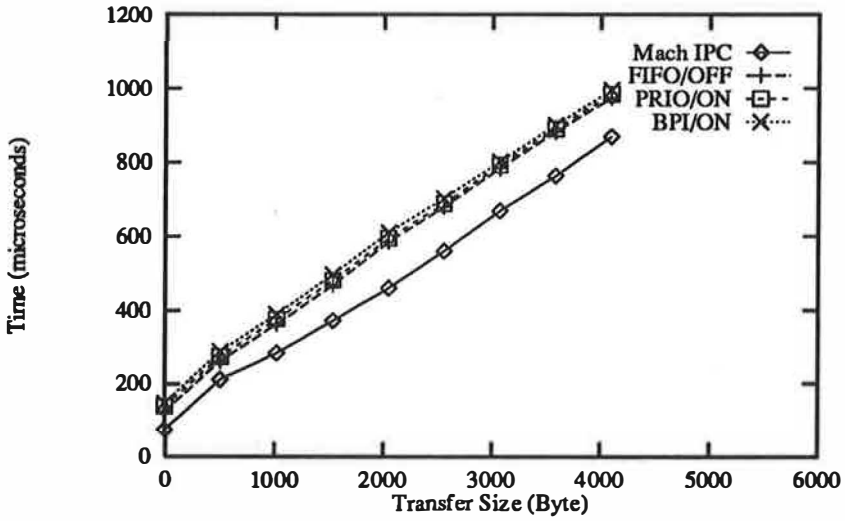


Figure 4: Round Trip Cost

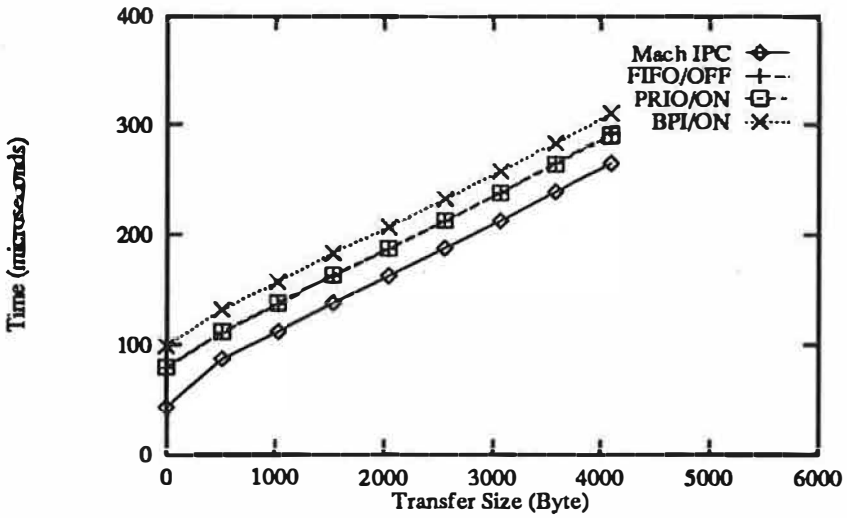


Figure 5: Message Send Cost when Receiver is Busy

5.2. Distributed Hartstone Benchmark

Distributed Hartstone (DHS)[7] is a series of extended Hartstone real-time benchmark programs and is designed for evaluating distributed real-time systems. Since the benchmark is designed for the distributed environment, the original DHS requires a set of servers and clients running on different machines connected by a network. In this evaluation, we executed the servers and clients on a single machine in order to evaluate the local IPC performance.

DSHcl is a Distributed, Synchronized, and Harmonic task set which originally tests the communication latency of the system. In this study, we expect this benchmark to evaluate the effect of priority assignment as well as communication latency. There are five client threads $\tau_1, \ldots, \tau_5$ which are periodic threads. Each of the client threads sends a request to the server before consuming its own workload. The benchmark uses the kilo-Whetstone as a unit of computation derived from the popular Whetstone benchmark. The server τ_{server} is an aperiodic thread. We vary the computation time of the server to gradually increase until any client thread misses a deadline. Figure 6 shows the structure of the benchmark. The timing requirements for each threads are shown in Table 1.

DHSpq is a Distributed, Synchronized, and Harmonic task set which is designed to test for priority queueing for communication packets. We expect this test to show the advantage of priority queueing over the conventional FIFO queueing. Like DSHcl benchmark, DSHpq has five clients and one server thread. The DSHqp benchmark is quite similar to the DSHcl except for the difference in granularity. Table 1 illustrates the timing requirement of the threads.

Table 2 summarizes the result of DHS benchmark. The numbers in the table represent the breakdown CPU utilization and the larger utilization number indicates the better system services.

The both results of the benchmarks show that RT-IPC provide higher CPU utilization comparing with Mach IPC. In case of *BPI/ON*, CPU utilization is slightly lower than that of *PRIO/ON*. This is caused by the overhead of priority inheritance.

6. Related Work

There has been a discussion and attempt of real-time IPC extension for multimedia systems on microkernel, however it was not realized before[17].

Many commercial and research operating systems provide some real-time features. For example,

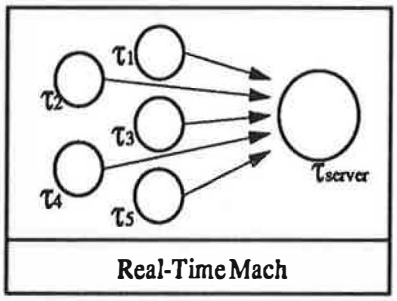


Figure 6: Distributed Hartstone Benchmark

Task	Workload	Period	
		DHScl	DHSpq
τ_1	1 KWS	80 ms	160 ms
τ_2	1 KWS	160 ms	320 ms
τ_3	2 KWS	320 ms	640 ms
τ_4	2 KWS	640 ms	1280 ms
τ_5	8 KWS	1280 ms	2560 ms
τ_{server}	variable	N/A	N/A

Table 1: DHS Task Set

	DSHcl	DHSpq
Mach IPC	60.6	55.3
FIFO/OFF	60.6	55.3
PRIO/ON	94.0	90.5
BPI/ON	93.4	88.3

Table 2: DHS result

Chorus[13] microkernel provides priority-based preemptive scheduling and fast interrupt latency. However, it does not provide the mechanisms to avoid unbounded priority inversion.

Lynx operating system[4] provides the priority inheritance protocol for real-time synchronization. However, the Lynx IPC does not provide priority inheritance for IPC, and cannot mix these two features without having priority inversions. POSIX[11] also proposed priority inheritance, but it does not discuss about the integration of IPC and synchronization.

There is a similar mechanism provided by QNX[5]. It provides priority-based message queueing and priority management mechanisms for IPC. However, it only supports synchronous communication mode, and is not integrated with the real-time synchronization facility.

A real-time extension of Mach for multimedia applications has been proposed[8]. It provides asynchronous event notification, preemptive deadline-driven scheduling, and temporal paging system. However, real-time extension of IPC nor Synchronization were not supported.

For real-time synchronization, the Stack Resource Policy (SRP)[1] has been proposed. It is also a starvation free locking protocol. It has been implemented only for real-time synchronization.

7. Conclusion

In this paper, we demonstrated the RT-IPC model, its implementation and its performance. Although the message transmission cost of RT-IPC is about 10 to 20% higher than the original Mach IPC, the DHS benchmark results indicate that RT-IPC significantly reduces priority inversions and improves the breakdown CPU utilization. We are still investigating the overhead of transmission cost and will reduce the gap between the Mach IPC and RT-IPC cost. Each policy has different cost and characteristic, a system designer needs to select a proper policy for the application program to minimize priority inversion problem and maximize CPU utilization.

RT-IPC has been used for Real-Time Server (RTS) and the Network Protocol Server (NPS)[9], and both servers successfully improve the breakdown CPU utilization of real-time applications.

Acknowledgments

We would like to thank the members of the ART Project and the Mach group for their valuable comments and inputs to the development of Real-Time Mach.

References

[1] T.P. Baker. A Stack-Based Resource Allocation Policy for Realtime Processes. In *Proceedings of IEEE 11th Real-Time System Symposium*, December 1990.

[2] R.P. Draves. A Revised IPC Interface. In *Proceedings of USENIX 1st Mach Workshop*, October 1990.

[3] R.P. Draves, B.N. Bershad, R.F. Rashid, and R.W. Dean. Using Continuations to Implement Thread Management and Communication in Operating System. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, October 1991.

[4] B.O. Gallmeister and C. Lanier. Early Experience with POSIX 1003.4 and POSIX 1003.4A. In *Proceedings of IEEE 12th Real-Time System Symposium*, December 1991.

[5] D. Hildebrand. An Architectural Overview of QNX. In *Proceedings of Workshop on Micro-Kernel and Other Kernel Architectures*, April 1992.

[6] K. Loepere. *Mach 3 Kernel Principles.* Open Software Foundation and Carnegie Mellon University, 1992.

[7] C. W. Mercer, Y. Ishikawa, and H. Tokuda. Distributed Hartstone: A Distributed Real-time Benchmark Suite. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, May 1990. Also available as Technical Report CMU-CS-90-110.

[8] J. Nakajima, M. Yazaki, and H. Matsumoto. Multimedia/Realtime Extension for Mach 3.0. In *Proceedings of Workshop on Micro-Kernel and Other Kernel Architectures*, April 1992.

[9] T. Nakajima, T. Kitayama, and H. Tokuda. Experiments with Real-Time Servers in Real-Time Mach. In *Proceedings of 3rd USENIX Mach Symposium*, April 1993.

[10] T. Nakajima, T.Kitayama, H. Arakawa, and H. Tokuda. Integrated Management of Priority Inversion in Real-Time Mach, 1993. The ART group Tech. Memo.

[11] *IEEE Standard P1003.4 (Real-time extensions to POSIX).* IEEE, 345 East 47th St., New York, NY 10017, 1991.

[12] R. Rajkumar. *Synchronization in Real-Time Systems: A Priority Inheritance Approach.* Kluwer Academic Publishers, 1991.

[13] M. Rozier, V Abrossimov, F. Armand, I Boule, M Gien, M Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser. Overview of the CHORUS Distributed Operating Systems Symposium, 1990. Chorus syste'mes, Technical Report CS-TR-90-25.

[14] S. Savage and H. Tokuda. RT-Mach Timers: Exporting Time to the User. In *Proceedings of USENIX 3rd Mach Symposium*, April 1993.

[15] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. Technical Report CMU-CS-87-181, Carnegie Mellon University, November 1987.

[16] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, September 1990.

[17] J.A. Test. Mach 3.0 Multimedia Real-Time Requirements, 1992. Proceedings of OSF Research Institute Symposium.

[18] H. Tokuda and T. Nakajima. Evaluation of Real-Time Synchronization in Real-Time Mach. In *Proceedings of 2nd USENIX Mach Workshop*, November 1991.

[19] H. Tokuda, T. Nakajima, and P. Rao. Real-Time Mach: Towards a Predictable Real-Time System. In *Proceedings of USENIX 1st Mach Workshop*, October 1990.

Takuro Kitayama is a Visiting Scientist in the School of Computer Science at Carnegie Mellon University. He is originally from Oki Electric Industry Co., Ltd. where he is employed since 1986. His research interests is real-time operating systems and multimedia operating systems. He received BS degree in engineering from Kougakuin University in 1986. He is a member of the ACM.

Tatsuo Nakajima is an Associate Professor of Information Center at Japan Advanced Institute of Science and Technology, where he is employed since 1993. Prior to that he worked on Real-Time Mach at Carnegie Mellon University. He received his PhD from Keio University in 1990 in distributed reliable computing. His research interest is operating systems for multimedia, high performance operating systems, reliable communications, and object-oriented languages. He is a member of the ACM and the Japan Society for Software Science and Technology.

Hideyuki Tokuda is a Senior Research Computer Scientist in the School of Computer Science at Carnegie Mellon and also an Associate Professor in the Faculty of Environmental Information at Keio University. His research interests include distributed real-time systems, multimedia systems, communication protocols, and massively parallel/distributed systems.

Tokuda received BS and MS degrees in engineering from Keio University and a PhD degree in computer science from University of Waterloo. He is a member of the IEEE, the ACM, the IPSJ, and the Japan Society for Software Science and Technology.

Availability

Mach 3.0 microkernel is free and available for anonymous FTP from cs.cmu.edu.

For RT-Mach, we distribute binaries for anonymous FTP from k.gp.cs.cmu.edu. The distribution includes the RT-Mach microkernel, libraries, and header files. The detailed description of the files are stored in /usr/arts/public/rtmach/README. You need to have an original Mach 3.0 environment before getting RT-Mach.

Fast Interrupt Priority Management in Operating System Kernels

Daniel Stodolsky
J. Bradley Chen
School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh PA 15213
danner@cs.cmu.edu
bchen@cs.cmu.edu

Brian N. Bershad
Department of Computer Science
and Engineering
University of Washington
Seattle WA 98195
bershad@cs.washington.edu

Abstract

In this paper we describe a new, low-overhead technique for manipulating processor interrupt state in an operating system kernel. Both uniprocessor and multiprocessor operating systems protect against uniprocessor deadlock and data corruption by selectively enabling and disabling interrupts during critical sections. This happens frequently during latency-critical activities such as IPC, scheduling, and memory management. Unfortunately, the cycle cost of modifying the interrupt mask has increased by an order of magnitude in recent processor architectures. In this paper we describe *optimistic interrupt protection*, a technique which substantially reduces the cost of interrupt masking by optimizing mask manipulation for the common case of no interrupts. We present results for the Mach 3.0 microkernel operating system, although the technique is applicable to other kernel architectures, both micro and monolithic, that rely on interrupts to manage devices.

1 Introduction

This paper describes a new technique, *optimistic interrupt protection*, that reduces interrupt management cost over conventional methods. While modern processor architectures have led to substantial overall performance improvements, operating systems have received significantly less benefit than application code [Anderson et al. 91, Chen & Bershad 93, Ousterhout 90]. One processor function that has not scaled well with processor speed is interrupt management. Operating systems use interrupts to control scheduling and I/O, and use interrupt masking to guarantee integrity of system resources shared across interrupt levels. This approach was efficient in many previous processor architectures (e.g, VAX), where the cost changing interrupt levels was small - generally less than ten instructions [Morse et al. 82, Levy & Eckhouse 89]. In modern architectures, however, interrupt masking may be up to an order of magnitude more expensive, contributing to poorer performance of system code.

This research was sponsored in part by The Advanced Research Projects Agency, Information Science and Technology Office, under the title "Research on Parallel Computing", ARPA Order No. 7330, issued by ARPA/CMO under Contract MDA972-90-C-0035, by the Xerox Corporation, and by Digital Equipment Corporation. Bershad was partially supported by a National Science Foundation Presidential Young Investigator Award.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of ARPA, XEROX, DEC, the NSF, or the U.S. government.

Machine	Processor	Instructions	Cycles
Luna88k	Motorola 88100 (25MHz)	68	96
Flamingo	Alpha 21064 (150MHz)	25	21
DECstation 5000/120	R3000 (20MHz)	65	80
DECstation 5000/200	R3000 (25MHz)	14	14
386 PC	Intel 386DX-25 (25MHz)	51	179
Gateway 66V	Intel 486DX2-66 (66MHz)	51	279

Table 1: *Overhead of changing the interrupt mask. Cycle counts are estimated, assuming no cache misses.*

Optimistic interrupt protection avoids the performance penalty of interrupt mask manipulation while preserving the semantics of the interrupt model. We have implemented optimistic interrupt protection in the Mach 3.0 microkernel for several different processor architectures. On the Omron Luna88k, we observed a 50% reduction in interrupt management overhead, resulting in a 5.3% speedup for interprocess communication.

The rest of this paper describes the technique and its performance. In Section 2 we review the basic problems introduced by interrupts, discuss the general model of interrupt handling into which optimistic interrupt protection fits, and motivate the need for a high performance mechanism. In Section 3 we describe the use and implementation of optimistic interrupt protection. In Section 4 we discuss the performance of our approach. In Section 5 we discuss related work. Finally, in Section 6 we present our conclusions.

2 Interrupt management

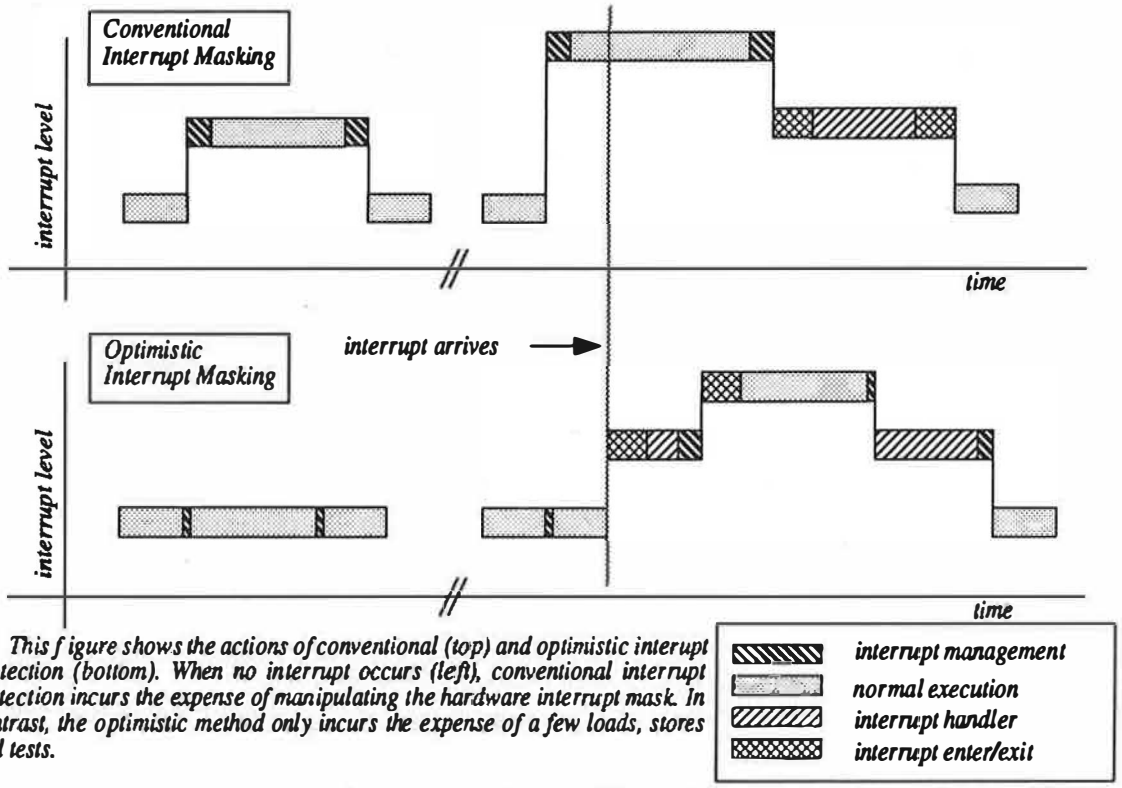
Operating systems generally rely on interrupts to respond to asynchronous events. Because interrupts introduce concurrency into the operating system kernel, system-level mechanisms are necessary to avoid deadlocks and protect system data structures from inadvertant concurrent accesses. *Interrupt masking* is a common technique for data protection in the presence of asynchronous events. Access to a potentially concurrent data is protected by setting the processor interrupt level to prevent all events that could potentially alter the data in question. Interrupt masking has been used successfully in a large number of operating systems, including Mach, Unix, VMS, and NT [Accetta et al. 86, Leffler et al. 89, Levy & Eckhouse 89, Custer 93]. It maps well onto a diverse array of hardware, from systems with a single interrupt level to processors with a rich interrupt structure [Bell et al. 82, Intel 90]. On a uniprocessor, no additional synchronization constructs are required. An important property of the interrupt masking model is that latency-sensitive events can preempt long-running low priority activities. Although alternatives to the interrupt model have been proposed [Cheriton 88, Massalin & Pu 89], simplicity, as well as the significant investment in existing system code and programmer experience provide significant economic incentives for preservation of interrupts as a model of system data protection.

Traditionally, interrupt masking has been efficient, requiring only a few cycles. Unfortunately, the time required to modify the hardware interrupt level has not scaled with processor speed improvements. In pipelined processors, writing the processor interrupt mask typically requires a pipeline flush [Motorola 90, DEC 92]. In superscalar systems, interrupt level manipulations require scalar instruction issue, further limiting performance [Sites 92]. Many recent RISC CPU implementations provide only a part of the interrupt mask logic on the processor package, with the remainder of interrupt masking implemented by off-processor hardware [Motorola 90, DEC 92]. For these systems, interrupt masking is a three step process: 1) disable processor interrupts, 2) write the off-chip mask register(s), and 3) finally re-enable processor interrupts. The first stage requires a pipeline flush, and the second stage requires a potentially expensive off-chip access. This represents a significant increase in the relative latency of interrupt mask manipulations. Table 1 shows the cost of a general interrupt mask raise/lower pair within the Mach 3.0 microkernel on a variety of architectures.

3 Optimistic interrupt protection

Optimistic interrupt protection exploits the fact that, in the common case, interrupts do not occur during critical sections. When a processor executing in the kernel enters a critical section, its sets a *software interrupt mask*, which

indicates the interrupts that need to be masked. The hardware interrupt mask is not changed. In the uncommon case that a lower-priority interrupt does occur, the interrupt handler prologue constructs an *interrupt continuation* (described below), updates the hardware interrupt mask as specified by the software interrupt mask, and returns control to the interrupted activity. Updating the hardware interrupt mask when the interrupt actually occurs prevents additional logically masked interrupts from occurring until the deferred handler has been executed. Though not strictly necessary, this tends to simplify the code. Moreover, it occurs after the interrupt, and is therefore off the anticipated fast path.



This figure shows the actions of conventional (top) and optimistic interupt protection (bottom). When no interrupt occurs (left), conventional interrupt protection incurs the expense of manipulating the hardware interrupt mask. In contrast, the optimistic method only incurs the expense of a few loads, stores and tests.

If an interrupt does occur (right), hardware masking defers the delivery of the interrupt until the end of the critical section in the conventional case. The interrupt is delivered promptly with optimistic interrupt protection, causing control to transfer to the interrupt handler. The interrupt handler recognizes this interrupt is logically masked, constructs an interrupt continuation, sets the hardware interrupt mask to the logical mask, and returns from the interrupt. Since the interrupt mask is raised, the critical section can run to completion without further interruption. When the critical section is done, the kernel discovers the presence of an interrupt continuation, resets the hardware interrupt mask, and executes the continuation. After the continuation is complete, the interrupt mask is cleared and normal processing resumes.

Figure 1: *Conventional and Optimistic Interrupt Protection*

An interrupt continuation is a data structure containing the state of the system at the time an interrupt is deferred. The interrupt continuation contains sufficient information to service the interrupt condition at a later time. The amount of information is typically quite small (e.g, the program counter and interrupt vector). At the end of the critical section, the processor checks for an interrupt continuation. Normally there is none, and processing continues following the critical section. If an interrupt continuation does exist, the processor handles the corresponding interrupt condition before resuming "normal" computation (see Figure 1). The interrupt continuation handles the deferred interrupt, restores the hardware interrupt mask to its original level, and returns to the normal execution stream.

As with traditional interrupt control, optimistic interrupt protection defers the execution of a masked interrupt handler until the end of the protected critical section. Unlike the traditional masking mechanisms, it requires that the (hardware and software) execution of the interrupt prologue code be both allowed and safe during protected sequences. As an example, if the interrupt prologue required a valid stack pointer, any code which places the stack pointer in an invalid state could not use optimistic interrupt protection. For the Mach 3.0 kernel, there are no such sequences on the Omron Luna88k, DECstation, or DEC Alpha.

In the optimistic case (the protected sequence runs without interruption), protection overhead is minimal. One

variable is set before the critical section, and at the end of the critical section that variable is reset and another variable (corresponding to the interrupt continuation) is checked. In the Omron Luna 88k implementation, this corresponds to two stores, one load and a test, all of which are executed by the processor at full speed. Not only is protection overhead small, it also scales with processor performance.

4 Performance

We have implemented optimistic interrupt protection in the Mach 3.0 kernel on the Omron Luna88k and Mips R3000 DECstation series. In both architectures, the interrupt continuation consisted of the register state at the time of the trap and a few additional words of state. Implementation took less than 3 days and and required no modification to assembler code routines. Table 2 shows the fast path overhead for interrupt management on these architectures. This sequence replaces the interrupt mask manipulations of Table 1. By using optimistic interrupt protection the length of the interrupt management path has been roughly halved.

Machine	Processor	Instructions	Cycles
Luna88k	Motorola 88100	51	51
DECstation 5000/120	R3000	31	31
DECstation 5000/200	R3000	31	31

Table 2: *Overhead of virtual interrupt mask manipulation. Cycle counts are estimated, assuming no cache misses. The Luna88k is a multiprocessor, so the virtual interrupt state is maintained on a per CPU basis. Most of the extra 20 cycles of overhead on the Luna88k are directly attributable to multiprocessor induced array indexing computations.*

To measure the impact of optimistic interrupt protection, we measured the performance of the Mach interprocess communication path. This path has already been highly optimized and contains only one interrupt protected critical section [Draves et al. 91]. Table 3 shows the performance of a cross address space null RPC with conventional and optimistic interrupt protection. The performance gain is larger than suggested by Tables 1 and 2 due to the idealized nature of those numbers. Both tables assume no TLB misses, cache misses, invalidation traffic or write buffer stalls; in practice, operating system code incurs a large contribution to cycles per instruction from all these factors [Chen & Bershad 93]. The reduction in path length and number of memory references in the interrupt management path therefore produces a greater than predicted benefit.

5 Related Work

One of the fundamental design decisions in an operating system is how to handle coordination between synchronous and asynchronous event handlers. Synchronous events happen within the context of the current execution stream (e.g, a system call), while a given asynchronous event can occur in the context of any instruction stream (e.g, I/O completion interrupts). Three approaches have been taken: interrupt masking as previously described, non-preemptable handlers, and lock-free synchronization.

Machine	Conventional	Optimistic	Speedup	Cycles saved
Luna88k	4400	4225	5.3%	175
DECstation 5000/120	2140	1840	14%	300
DECstation 5000/200	1234	1198	2.9%	36

Table 3: *IPC performance. Shown are the cycles for a null RPC with optimistic and conventional interrupt management. One cycle on the Luna88k is 40 nanoseconds, so IPC latency is reduced by 7 microseconds from 176 to 169. The cycle times on the 5000/120 and 5000/200 are 50 and 40 nanoseconds respectively.*

In the non-preemptable approach, both synchronous and asynchronous event handlers run uninterruptably to completion. The V kernel and many real time systems follow this approach [Berglund 86, Stankovic & Ramamritham 88]. Unfortunately, non-preemptable interrupt handlers impose serious constraints on handler structure: all handlers must be short to ensure that the latency of high priority events is low, and handlers cannot containing blocking operations (e.g. device status register polling). While this approach can lead to a high performance operating systems, difficulties inherent in this code style have prevented its widespread use.

Recent research has demonstrated the use of highly concurrent lock-free data structures [Herlihy 90, Wing & Gong 92]. A system using lock-free synchronization can be free from data corruption, deadlock and priority inversion even in the case of interrupts [Massalin & Pu 91]. In addition, lock-free data structures provide the necessary synchronization for both multiprocessors and non-preemptive execution. Consequently, lock-free data structures suggest an attractive approach for structuring operating systems. Unfortunately, lock-free data structures can require special synchronization hardware that is neither generally available nor inexpensive [Herlihy 91, Motorola 90][1]. Recently, researchers have proposed architectural modifications to efficiently support lock-free operations [Herlihy & Moss 93].

The division of synchronization mechanisms into an inexpensive optimistic and (relatively more) expensive pessimistic case has been applied elsewhere. Restartable atomic sequences offers a mechanism for constructing efficient user-level synchronization primitives in a preemptively scheduled environment [Bershad et al. 92].

6 Conclusions

Optimistic interrupt protection is an application of optimistic synchronization to interrupt priority management in operating system kernels. It provides the same semantics as traditional interrupt management with much less overhead. A measurable speedup of the IPC path in the Mach 3.0 microkernel was obtained by using this technique. The method is applicable to any kernel that uses interrupt masking to guarantee data integrity.

References

[Accetta et al. 86] Accetta, M. J., Baron, R. V., Bolosky, W., Golub, D. B., Rashid, R. F., Tevanian, Jr., A., and Young, M. W. Mach: A New Kernel Foundation for Unix Development. In *Proceedings of the Summer 1986 USENIX Conference*, pages 93–113, July 1986.

[Anderson et al. 91] Anderson, T. E., Levy, H. M., Bershad, B. N., and Lazowska, E. D. The Interaction of Architecture and Operating System Design. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 108–121, April 1991.

[Bell et al. 82] Bell, C. G., Newell, A., and Siewiorek, D. P. *Computer Structures:Principles and Examples*, pages 110–128. McGraw-Hill, 1982.

[Berglund 86] Berglund, E. J. An Introduction to the V-System. *IEEE Micro*, 10(8):35–52, August 1986.

[Bershad et al. 92] Bershad, B. N., Redell, D. D., and Ellis, J. R. Fast Mutual Exclusion for Uniprocessors. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1992.

[Chen & Bershad 93] Chen, J. B. and Bershad, B. N. The Impact of Operating System Structure on Memory System Performance. In *Proceedings of the 14th ACM Symposium on Operating System Principles*, December 1993. To appear.

[Cheriton 88] Cheriton, D. R. The V Distributed System. *Commun. of the ACM*, 31:314–333, March 1988.

[Custer 93] Custer, H. *Inside Windows NT*. Microsoft Press, Redmond, WA, 1993.

[DEC 92] DEC. *DECChip 21064-AA RISC Microprocessor Preliminary Data Sheet*. Digital Press, Manyard, MA, 1992.

[Draves et al. 91] Draves, R. P., Bershad, B. N., Rashid, R. F., and Dean, R. W. Using Continuations to Implement Thread Management and Communication in Operating Systems. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 122–136, October 1991.

[Herlihy & Moss 93] Herlihy, M. and Moss, E. Transactional Memory - Architectural Support for Lock Free Data Structures. In *The 20th Annual International Symposium on Computer Architecture*, May 1993.

[1] With only minimal hardware support (e.g, load linked / store conditional), implementation of nontrivial data structures (doubly linked lists, trees) requires substantial data copying. In addition, when these primitives have been implemented, their performance is often quite poor (alpha L_l/S_c) - often as long as an uncached load and store [DEC 92].

[Herlihy 90] Herlihy, M. A Methodology for Implementing Highly Concurrent Data Structures. In *Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 197–206, March 1990.

[Herlihy 91] Herlihy, P. M. Wait-free synchronization. *ACM Transcations on Programming Languages and Systems*, 13(1):124(26), January 1991.

[Intel 90] Intel. *i486 Microprocessor Programmer's Reference Manual.* Intel, Mt. Prospect, IL, 1990.

[Leffler et al. 89] Leffler, S. J., McKusick, M., Karels, M., and Quarterman, J. *The Design and Implementation of the 4.3BSD UNIX Operating System.* Addison-Wesley, 1989.

[Levy & Eckhouse 89] Levy, H. M. and Eckhouse, R. H. *Computer Programming and Architecture: The VAX-11 (2nd Edition).* Digital Press, Bedford, MA, 1989.

[Massalin & Pu 89] Massalin, H. and Pu, C. Threads and Input/Output in the Synthesis Kernel. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 191–201, December 1989.

[Massalin & Pu 91] Massalin, H. and Pu, C. A Lock-Free Multiprocessor OS Kernel. Technical Report CUCS-005-91, Department of Computer Science, Columbia University, 1991.

[Morse et al. 82] Morse, S. P., Ravenel, B. W., Mazor, S., and Pohlman, W. B. *Computer Structures: Principles and Examples*, pages 615–646. McGraw-Hill, 1982.

[Motorola 90] Motorola. *MC88100 RISC Microprocessor User's Manual.* Prentice Hall, Englewood Cliffs, NJ, 1990.

[Ousterhout 90] Ousterhout, J. K. Why Operating Systems Aren't Getting Faster As Fast As Hardware. In *Proceedings of the Summer 1991 USENIX Conference*, pages 247–256, June 1990.

[Sites 92] Sites, R. L., editor. *Alpha Architecture Reference Manual.* Digital Press, 1992.

[Stankovic & Ramamritham 88] Stankovic, J. and Ramamritham, K. A New Hard Real-Time Kernel. In *Hard Real-Time Systems*, pages 361–370. IEEE, 1988.

[Wing & Gong 92] Wing, J. M. and Gong, C. A Library of Concurrent Objects. CMU CS TR 90-151, School of Computer Science, Carnegie Mellon University, 1992.

User Level IPC and Device Management in the Raven Kernel

D. Stuart Ritchie and Gerald W. Neufeld
{sritchie,neufeld}@cs.ubc.ca

Department of Computer Science
University of British Columbia
Vancouver, B.C. V6T 1Z2
Canada

Abstract

The increasing bandwidth of networks and storage devices in recent years has placed greater emphasis on the performance of low level operating system services. Data must be delivered between hardware devices and user applications in an efficient matter. Motivated by the need for low overhead operating system services, the Raven kernel utilizes user level implementation techniques to reduce kernel intervention for many common services. In particular, our user level send/receive/reply communication implementation generates no kernel interactions per iteration in the best case, and two kernel interactions in the worst case. In more general cases, we observe approximately one kernel interaction for every two send/receive/reply iterations. Device driver support is also done entirely at the user level reducing copy costs and context switching.

1 Introduction

The speed of network channels and storage devices has increased by an order of magnitude in recent years (10Mbps Ethernet to 100Mbps FDDI and 140Mbps ATM). This increased bandwidth places additional burden on the operating system to deliver data between hardware devices and user applications. In order to sustain such speeds, device drivers must be invoked with low latency and be able to communicate high volumes of data to and from applications. We believe that pure kernel mediated architectures such as Mach [ABB+86] and V [Che88], even optimized using continuations [DBRD91], involve significant overhead.

Motivated by the need for low overhead operating system services in high speed protocol processing applications, we have implemented a lightweight kernel for shared memory multiprocessors. Threads, IPC, and device management are implemented at the user level, while task and virtual memory management are implemented in the supervisor kernel. Existing systems such as URPC [BALL90], and the "continuous media" system [GA91] demonstrate the viability of this approach. Our current implementation is based on the Motorola MVME188 Hypermodule, a 25MHz quad-processor 88100 machine [Gro90].

Our goal with the Raven kernel is to provide a lightweight environment for multithreaded parallel applications. The applications that we are currently investigating are high-speed networking and disk activities. In an environment based on threaded parallelism and high device interrupt rates, a great deal of context switching is to be expected when running such applications. We have designed and implemented our system accordingly.

By moving several of the high-use kernel services into user space, less time is spent invoking operations. The general motivation is to reduce the overall number of user/kernel interactions. Several techniques are employed by Raven to do this:

- User level thread scheduling. Rather scheduling threads in the kernel, move the scheduling code into the user space.
- User level interrupt handling. Allow interrupt handlers to upcall directly into the user space. Device drivers can be implemented completely in user space, eliminating the costs of moving device data between the user and kernel.
- User level interprocess communication. By making extensive use of shared memory between client and server address spaces, data copying through the kernel is eliminated.
- Low level synchronization primitives. Provide a simple mechanism to allow an event to be passed from one address space to another. With appropriate hardware, remote processor interrupts can be implemented completely at the user level.

This paper discusses the design and performance of our user level IPC implementation and device driver management. We introduce the design of the overall system, and then discuss the IPC and device management facilities and how they interact. The paper follows with a performance evaluation of our current implementation.

2 Overall kernel design

The Raven kernel is small, lightweight microkernel operating system for shared memory multiprocessors consisting of a supervisor kernel and user level library. The supervisor kernel and user level library (Figure 1 compiles into 52K of executable code and 72KB of data. The size and scope of the Raven kernel is roughly similar to the QNX [Hil92] operating system: the supervisor kernel provides a simple set of abstractions, from which a set of more complex services may be constructed. Unlike QNX, however, the Raven kernel takes advantage of symmetric multiprocessing and user level design techniques.

Two main abstractions are provided by the supervisor kernel: tasks and virtual memory. Task creation, destruction, and scheduling (address space switching), are implemented inside the kernel. A task consists of an address space that is scheduled for execution by the kernel on one or more processors. The kernel also maintains strict control over page tables and free memory lists, so all virtual memory allocations and mapping are provided by system calls.

All other services are provided by the user level: threads, semaphore synchronization, interprocess communication, and device management. These features are accessible to application programs via inline macros or procedure calls, rather than more expensive kernel traps. Threads are preemptively scheduled from processor to processor in an effort to balance work load. Semaphores can be used to coordinate threads in local and remote address spaces. Extensive use of shared memory allows disjoint address spaces to efficiently communicate scheduling information and interprocess communication data. Figure 1 shows the supervisor kernel in relation with the user level library.

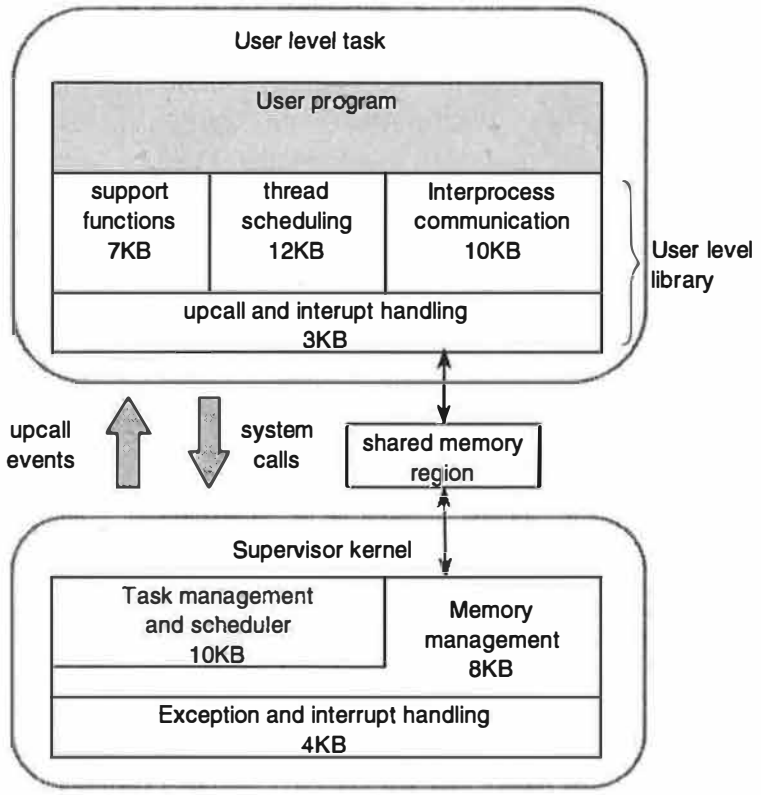


Figure 1: High level system organization.

The user level thread scheduler communicates and coordinates scheduling activities with the supervisor kernel using a per-task shared data structure and system calls. The shared data structure maintains several fields which allow the user level scheduler and supervisor kernel to make scheduling decisions without crossing user/kernel address space boundaries. Under ideal conditions, threads can migrate amongst processors without kernel intervention. However, certain cases exist where kernel assistance is required. One such case occurs when an address space switch is required.

The MVME188 hardware provides a simple and efficient means to deliver interrupts to remote processors in the system. We use this feature throughout the system to aid in the thread and task scheduling decisions required for IPC interactions and thread/task management. For example, when a new thread is created, an interrupt is delivered to the next available idle processor to run that thread. Similarly, invoking the IPC mechanism to send a message results in an interrupt delivered to the processor that is currently executing the desired destination task. Idle processors never scan lists searching for work to do – work is delivered to idle processors in the form of interrupt notifications.

Built on top of this interrupt mechanism is an asynchronous task signalling facility. The task signalling facility provides a mechanism to asynchronously send signal messages from one task to another. This software implementation is roughly similar to Cheriton's hardware work with "address-valued signals" [CK93]. A signal message is a simple two word structure which identifies the signal type and message data. Each user level scheduler maintains a FIFO queue to store signal messages, and implements a set of signal handlers, one for each type. Signal handlers are invoked on the receipt of each signal message. Task

signals are the the basis for IPC and semaphore operation throughout the kernel.

A task is delivered a signal by a non-blocking user level library function, `task_signal()`. This function avoids system calls on the local processor by using the remote software interrupt facility to notify destination tasks of the arrival of signal messages. There are three possible ways that signals are delivered:

1. If the destination task is currently executing on a remote processor, an interrupt is issued to that processor.
2. If the destination task is not executing on a remote processor, then issue an interrupt to an idle processor.
3. If there are no idle processors and the destination task is not active, perform a system call to the local processor to initiate a task switch.

Only the last item (3) produces an explicit system call by the local processor. The other items avoid a system call locally, allowing the processor to proceed with its own work. Work is distributed to remote processors in this fashion.

3 Interprocess communication

The user level IPC library provides a port-based synchronous send/receive/reply[1] interface as well as asynchronous send/receive. Both of these interfaces utilize user level shared memory and the task signalling facility described above to reduce the frequency of kernel interactions. Systems such as Mach and Chorus, extend their IPC models across the network. We only consider the inter-machine case.

The port-based approach to interprocess communication uses a port number as the mailbox address for reliable message delivery. Rather than implementing send/receive/reply in terms of two send/receive ports, we have implemented the two models separately. There are several reasons for doing this, one being performance.

Figure 2 shows a high level view of how the main IPC data structures are organized. Two port descriptor tables are allocated by the initial task at boot time, one maintaining entries for synchronous ports, the other maintaining entries for asynchronous ports. The tables are shared amongst all user level tasks in the system. A port descriptor contains information pertaining to the state of the port queue, and a set of semaphores to coordinate client and server operations. At port creation time, the IPC library searches the appropriate table for a free entry and allocates an associated FIFO message queue. The number of messages in this queue and their size is specified at port creation time. The send/receive/reply implementation uses the same queue for storing both send and reply messages – a second queue is not required. After a port is successfully created, the resulting descriptor index is used throughout the system as a port identifier.

The port's message queue is shared in a region of virtual memory between the server and each of the clients involved in the communication. Clients wishing to communicate to a server over a port must initially "establish a connection" to the server by calling a library function that performs the message queue mapping and registration functions.

[1] Our send/receive/reply implementation permits the reply stage to be deferred until a later time, out of sequence with the arrival of messages similar to the V System[Che88].

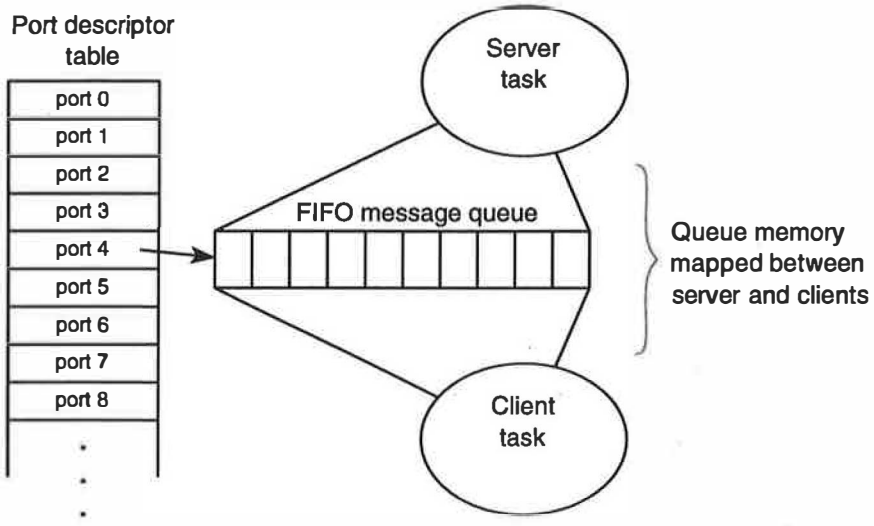


Figure 2: Port descriptor table and message queue mapped into a client and server.

3.1 IPC algorithm

The work performed by an interprocess communication facility can be divided into two parts. The first part manages the movement of message data between the sender and receiver, usually by means of a message queue. Traditional kernel based systems maintain message queues inside the kernel: message data must be marshaled in and out of the kernel on each invocation. Raven's exposure of message buffers to user space allows user code to directly marshal message data. This can save a data copy operation on the receiver side because the receiver has direct access to the message buffers. A second "user level" copy need not be made.

Once message data is queued for delivery, the second part of an IPC transaction involves notification to the recipient that a message has arrived. The notification step passes control to tasks where blocked recipient threads wait, often involving a processor allocation decision (a client task may be switched out so that a server task can run). For example, when a client thread sends a message to a server, the server must be notified. In traditional kernel based systems, this notification step normally requires kernel support. In the Raven kernel, the task signalling mechanism is used as the low level notification system. As noted above, this mechanism avoids kernel intervention in two out of three cases.

Synchronous send/receive/reply transactions are inherently more complex than asynchronous send/receive. The synchronous case requires the client senders to block and wait for a reply from the servers. In the best case, no notifications are required for either synchronous or asynchronous IPC. In the worst case, two notification steps are required for synchronous IPC: the client notifies the server that a message awaits, and the server notifies the client that a reply awaits. For the asynchronous case, only one notification is required: the client notifies the server that a message awaits.

In best case, the IPC library can use properties of the FIFO message queues and multiprocessing to avoid the notification step, and ultimately avoid kernel intervention altogether. Looking at the simpler asynchronous case, we can see how this is accomplished:

1. If a server thread is not currently blocked waiting for a message when a client performs a send, no notification by the client is necessary. When the server thread eventually performs a receive operation, it will notice a message waiting and immediately pull it from the queue without blocking.

2. A non-empty message queue indicates to the client that the server has already been notified that a message awaits. Thus, notification to a server only occurs when a server thread is blocked and does not already have messages waiting for it.

3. However, if a message queue is full, and cannot accept additional client messages, the sending client must block. As the server eventually empties the message queue, it must send notification to the blocked clients indicating that more room exists in the queue.

The use of these properties are summarized by the following algorithms used to implement the asynchronous send and receive primitives. We first examine the send primitive. The send primitive generates at most one notification per invocation:

1. If there are no free message buffers available, block on a a free-list semaphore.

2. Claim the next free message buffer and copy in the message.

3. Deliver a notification signal to the destination server task only if the send queue was previously empty and a server thread is blocked. Otherwise, do not send a notification signal.

The receive primitive generates at most one notification per invocation also:

1. Block the calling thread if there are no messages waiting. Continue otherwise.

2. Dequeue the next message and copy it into the calling thread's buffer.

3. Return the message buffer to the queue and increment the free-list semaphore. If a client is blocked waiting for a free message buffer, deliver a notification signal to the client, indicating that a free message buffer exists.

As noted above, the send/receive/reply mechanism is more complex than send/receive because the sender must always block and wait for a reply. This can result in two notification signals per interaction in the worst case. However, as seen in the asynchronous send analysis above, it is possible to eliminate notification signals. We can apply the same principles to both the send stage and reply stage. This allows send/receive/reply to operate without notification (and therefore without kernel intervention) in the best case. The algorithms for the send/receive/reply primitives are summarized below.

The send operation requires at most one system call per invocation, or possibly a task signal invocation:

1. If there are no free message buffers available, block on a a free-list semaphore.

2. Copy the user's message into the buffer.

3. If the send queue is empty, and a server thread is blocked waiting for a message, deliver a notification signal to the destination server indicating that a message awaits.

4. Block the sending thread and wait for a reply.

5. Wake up when the reply comes and copy the reply message to the user's buffer.

6. Return the message buffer to the queue and increment the free-list semaphore. If a client is blocked waiting for a free message buffer, deliver a notification signal to the client, indicating that a free message buffer exists.

The receive operation does not require any kernel interaction at all.

1. If a message awaits in the port queue, return a pointer to the received message buffer to the user.

2. Otherwise, if the port queue is empty, block the calling thread and wait for a message.

3. Wake up when a message arrives and return a pointer to the message buffer to the user.

The reply operation requires a notification signal only when there are no reply messages queued for the client task. This case is analogous to the sender's situation, except that here a reply is being delivered:

1. Enqueue the reply message.

2. If there are existing reply messages in the queue for the client, simply return.

3. Otherwise, deliver a notification signal to the client to indicate that a reply awaits.

It should be noted that all access a FIFO queues and port descriptors require synchronization. This is accomplished using spin-locks. Spin-locks are efficiently implemented by utilizing the Motorola 88200 cache coherency mechanism. No kernel intervention is required to implement the locks.

4 Device driver management

Hardware device drivers are implemented completely in user space. User level applications register hardware device handlers through the kernel interrupt dispatch mechanism and map in the device registers to their address space. When a device interrupt occurs, an upcall [Cla85] is issued to the task which contains a handler for the interrupt. The handler is executed to satisfy the device and processing resumes.

Each interrupt generated by a device causes an upcall into user space. On the surface, the additional cost of traversing from kernel to user space compared to a pure kernel interrupt handler would appear prohibitive. However, we observe the following advantages with this technique:

- There is no need to copy or map data between the kernel and user level.

- Execution can continue in user space after processing the interrupt. A kernel level handler eventually requires an upcall into user space to allow applications to process data.

- Device drivers can be dynamically loaded and unloaded without the complexity of dynamic linking.
- Device drivers can be implemented directly in the application that uses the device, reducing communication costs and latency.

4.1 Interrupt management

The MVME188 interrupt management hardware is fully symmetric. Any processor in the system can respond to any particular interrupt by setting appropriate bits in the processor's interrupt enable register. In our system with four processors, there are four interrupt enable registers. The kernel manages these registers in an effort to position device interrupts to minimize invocation latency of user level interrupt handlers. The interrupt enable bits follow the migration of their associated application programs.

For example, the Ethernet interrupt handler function is implemented within a user level task. An upcall event is dispatched into this task to execute the handler whenever the Ethernet interrupt occurs. If the Ethernet task address space is not enabled on the processor where the interrupt occurs, the task must be switched in. If the interrupted processor is currently running a different task, then it must be switched out before the Ethernet task can be switched in. This sequence of steps involve address space changes and data structure manipulation that greatly increases interrupt handler latency. The kernel attempts to avoid this situation by positioning interrupt enable bits on processors that are currently executing the associated task.

This interrupt management scheme also allows devices to operate in parallel. For example, one processor can enable its Ethernet bit and another processor can enable its disk or serial port bit. Figure 3 demonstrates this distribution of interrupt handling chores. Processor 1 is currently running the file system server, and thus has the SCSI disk interrupt set. Processor 2 and 3 are running the TCP networking software, and thus are sharing the serial port and Ethernet interrupt.

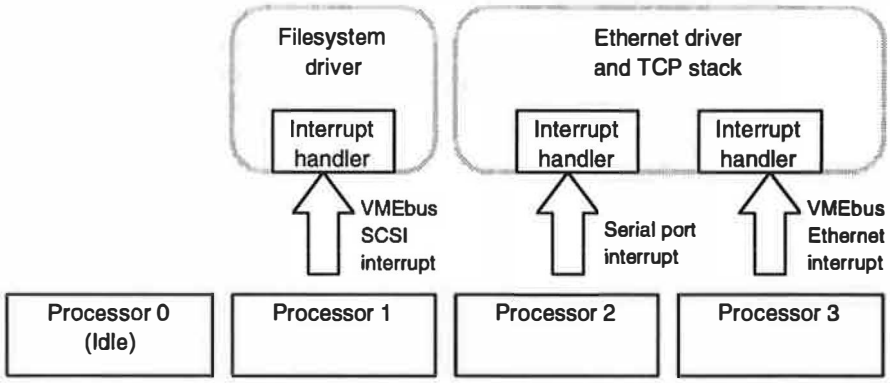


Figure 3: Interrupt management across three busy processors.

4.2 User level preemption

Interrupts preempt user level threads and cause scheduling events to occur. For example, if a timer interrupt occurs, an upcall event is sent to the user level to indicate that it's time to schedule the next ready thread. In a multiprocessor environment, special care must be taken to ensure that the rescheduling of threads in the presence of spin-locks does not adversely affect performance. A naive approach would allow interrupts to occur at any point during user level execution. This can result in very poor performance if threads are using spin-locks for concurrency protection. If a thread is preempted while holding a spin lock, then all other threads that try to access the lock must wait until the original thread is rescheduled and releases its lock. The original thread may not be rescheduled for some time, causing all other threads to waste CPU time, uselessly spinning.

The solution implemented in the Raven kernel involves close participation between the user level and kernel. Whenever the user level acquires a spin lock, a global `lock_count` variable is incremented. Whenever an interrupt occurs that would cause an upcall event into user space, the interrupt handler checks the `lock_count` variable. A non-zero value indicates that a critical section is currently being executed, and control must be returned to the critical section. Before the interrupt handler restores user registers and returns control to the critical section, it sets the `upcall_pending` variable to indicate that an interrupt occurred. When the user level regains control and finishes its critical section, the `upcall_pending` variable is checked, and if set, the thread will save its context and handle the original reason for preemption.

As implemented on the 88100, the two variables `lock_count` and `upcall_pending` are not stored as conventional variables at all. Rather, each of them share the processor's `r28` register. This is done to ensure that access to the variables is atomic. For example, to increment `lock_count`, a single `addu r28,r28,1` instruction is performed. If `lock_count` were a conventional variable, then incrementing it would require the use of a spin lock – which would be recursive, since `lock_count` itself is used within the locking routines.

4.3 Dispatching interrupts

Dispatching interrupts in the Raven kernel is a two-level process. Interrupt dispatchers are implemented at both the supervisor and user level. When a device interrupt occurs on a processor, the register context is saved into the user's thread control block, and the supervisor kernel interrupt dispatcher is called to begin processing the interrupt. If the interrupt is intended for a user level handler, an upcall is generated into the appropriate task where the handler is implemented, and the user level dispatcher takes over. The user level dispatcher calls the appropriate handler. Figure 4 shows the interrupt execution path from the supervisor dispatcher to the interrupt handlers.

The supervisor interrupt dispatcher maintains a table of all the possible interrupt sources and the location of the handler functions. Some handler functions are implemented directly in the kernel, such as the system clock tick. To handle a kernel level interrupt service routine, the dispatcher simply makes a function call to the service routine.

For user level handlers, the procedure is more complicated. The supervisor dispatcher checks the appropriate interrupt entry, and if the currently executing task is registered to handle that interrupt, an interrupt upcall event is delivered to the task. The particular interrupt vector bit that caused the interrupt is passed with the upcall event so the user

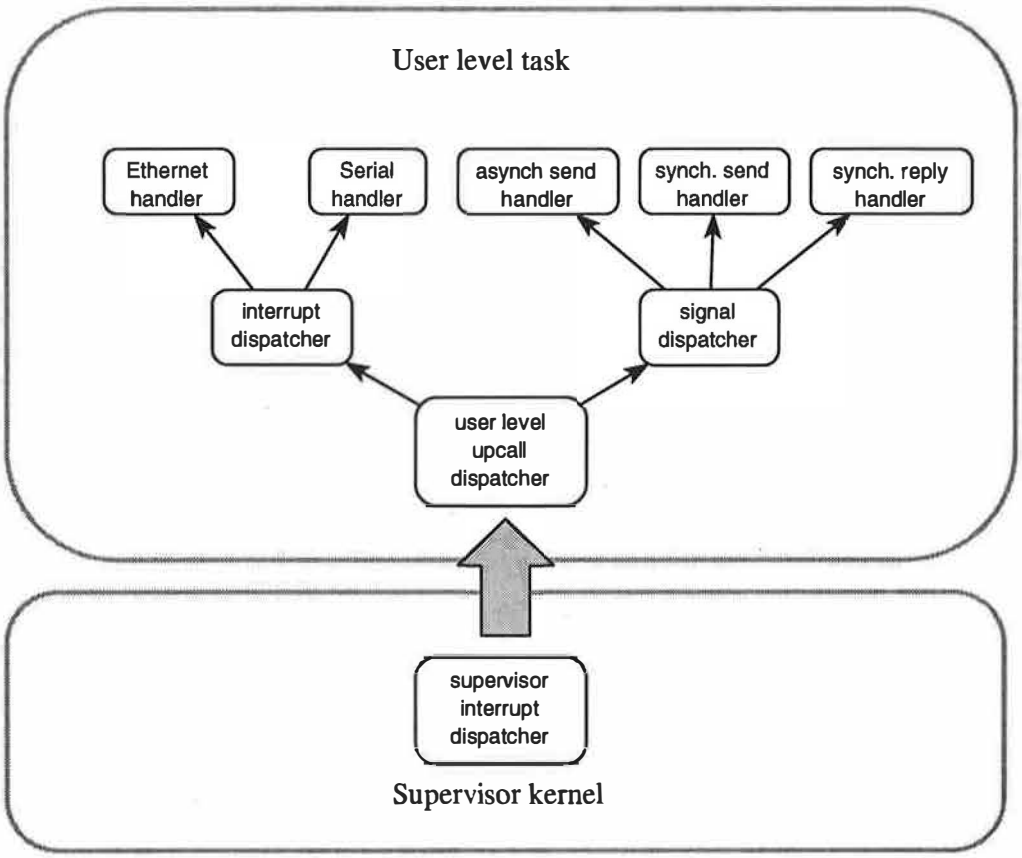


Figure 4: Interrupt execution path from supervisor dispatcher to the user level, including two interrupt handlers and IPC signal handlers.

level dispatcher can identify the proper handler.

If the currently executing task does not handle the interrupt, but another task does, then the current task must be switched out and the interrupt handling task must be switched in. Before the current task is placed back onto the task ready queue, the thread which was interrupted must be cleaned up so that it can be rescheduled. While the register context for the thread has been properly saved by the initial interrupt trap function, the user level thread kernel must be notified that one of its threads has been preempted, so the thread can be placed back on the user level thread ready queue. To do this, the current task is issued an upcall event so that the thread scheduler can place the interrupted thread back onto the thread ready queue. The thread scheduler then immediately returns to the kernel, where the interrupt handling task is finally activated.

4.4 Handler functions and device access

Before an interrupt handler can be invoked, the device register set must be mapped into the application address space and the handler function must be registered with the kernel. This is done during the initialization phase of the device driver. When the device driver terminates, the handler function must be removed and the device register set must be unmapped from the application.

The virtual memory module exports a function to the user level called `vm_map_device()` which allows applications to map device registers into user space. At initialization time, the device driver supplies the physical address and size of the device registers, and a virtual address is returned pointing to the mapped device registers. The function `vm_unmap_device()` allows the device driver to later remove the mapped memory from its address space.

A user level library routine manages the registration of interrupt handlers for device drivers. The caller specifies the function address and the associated interrupt vector. The registration routine then performs a system call into the kernel to notify the supervisor dispatcher of the new interrupt handler, and the appropriate interrupt enable bit for specified device is set. Once this is done, interrupts occurring on the device will be vectored to the user level handler.

User level interrupt handlers are executed within a controlled environment. Preemption is disabled during this time, so the handler is guaranteed uninterrupted access to the device registers. However, handler must not spend more time than is necessary to service the interrupt, or time will be taken away from other system activities. Additionally, the handler must be careful not to execute any library functions that perform thread or task context switching.

Interrupt handler functions typically coordinate their events with user threads using a semaphore and a shared data structure. When an event occurs in the handler that must be communicated to a thread, the event is recorded and the semaphore is signalled. A thread blocked on this semaphore will awake and be able to examine the event information. When a handler function is finished accessing the device, it returns to the dispatch routine where the thread scheduler takes over.

5 Performance

This section presents some simple benchmark results to demonstrate our implementation. We have constructed several test cases that exercise the interprocess communication primitives and interrupt handling features. To summarize:

- A single send/receive/reply interaction occurring between two tasks with no kernel interactions can complete in 42 microseconds. This represents the best case scenario.
- Average send/receive/reply times between several communicating threads over a period of time is measured at 90 microseconds. This represents approximately one kernel interaction per transaction.
- Worst case send/receive/reply of 145 microseconds is achieved by limiting two communicating threads to a single processor.
- User level interrupt handler invocation latency is 14.0 microseconds.

5.1 Interprocess communication

This section measures the performance throughput of the IPC library. The benchmark combines many of the primitive system services: remote interrupt dispatching, task signal notification, semaphores, and task and thread scheduling.

	1 CPU	2 CPU	3 CPU	4 CPU
1-send/1-recv/reply	145 usec	105 usec	105 usec	105 usec
2-send/2-recv/reply	108	95.3	90.3	89.6
10-send/10-recv/reply	89.3	92.5	94.9	95.6

Table 1: IPC performance for synchronous ports, 4 byte data message, 20 element message queue.

The global IPC test cases create two address spaces: a server task, and a client task. The server task allocates a port with a queue containing 20 message buffers. A number of server threads are created to listen for messages on the port. The client task creates a number of client threads and bombards the server with messages.

5.1.1 Synchronous IPC performance

Table 1 contains performance results for synchronous IPC using various combinations of processors and threads. The simple case of 1-send thread and 1-receive thread on a single processor demonstrates the worst case performance of 145 microseconds per interaction. Each iteration requires two address space changes. This performance is improved upon by the addition of multiple processors. In the two processor case, the client and server reside on different processors and can therefore avoid address space switching. Instead, notification signals are delivered via the remote interrupt mechanism. Additional processors do not help improve this case because there is no parallel computation involved.

Synchronous IPC performance increases slightly when more client and server threads are added. In the uniprocessor case, this is because clients and servers can copy messages in and out of the queues at once without switching tasks. For example, the 10 client threads can each copy their messages into the send queue before the task switches. Likewise, the 10 server threads can stack up their reply messages. This has the effect of reducing the overall number of task switches per IPC interaction.

An interesting case appears when 10 client threads and 10 server threads communicate using multiple processors. The IPC interactions actually get slower. We believe that this is a combination of spin-lock contention and poor scheduling decisions by the task and thread schedulers. Rather than balance the client and server threads on an even number of processors, the schedulers position tasks and threads naively. Thus a great deal of task switching results as client and server threads rapidly migrate amongst processors. A better scheduler might be able to recognize communicating tasks and position them on processors accordingly.

The figures in Table 1 are averages over large number of iterations. We modified the dual processor test case above to measure the quickest and slowest send/receive/reply transaction out of all these iterations as 42 microseconds and 120 microseconds, respectively. We believe this discrepancy to be caused by the overlapping of execution within the IPC primitives. The quickest iteration occurs when the client sends a message exactly at the same time as the server enters the receive primitive.

	1 CPU	2 CPU	3 CPU	4 CPU
1-send/1-recv	43.2 usec	33.9 usec	32.1 usec	32.0 usec
2-send/2-recv	44.5	32.2	31.8	31.0
10-send/10-recv	44.3	33.9	32.0	32.5

Table 2: IPC performance for asynchronous ports, 4 byte data message, 20 element message queue.

5.1.2 Asynchronous IPC performance

Table 2 summarizes the results for several asynchronous IPC test cases. The asynchronous message transfers are much faster overall because of the reduced context switching requirements between the client and server. Client threads have no problem keeping the port queue full of data for the server threads. A client can simply loop forever, assuming the server keeps up.

Increasing the number of threads results in a slight performance hit. We believe this is due to the increased number of thread descriptors being managed by the thread scheduler. The thread scheduler is very simplistic, and may position threads on processors in a non-optimal fashion. Also, increasing the number of threads increases the number of stacks and data structures to manage while context switching, possibly affecting cache performance.

As seen in the synchronous case, performance improvements decline as more processors are added to the system. We believe the reason for this is spin-lock contention. The workload performed by the client and server threads is null, so all their effort is spent trying to access shared data structures such as port descriptors and queue. If the client and servers performed some amount of work, as in a real application, most of their time would be spent outside of the IPC primitives, leaving less opportunity for lock contention.

5.2 Interrupt handling performance

The performance of interrupt handling is a critical concern for high speed device drivers and scheduling performance. Interrupt handling must be as lightweight as possible to ensure low latency dispatch times to device drivers. Interrupt handling and dispatching in a monolithic kernel is fairly straightforward: trap the interrupt, save context, and call the interrupt service routine. In the Raven kernel, since device drivers are implemented at the user level, device interrupts must take the journey up into the user level for processing. However, once at the user level, execution can continue with application processing.

An experiment was constructed to measure the execution latency time to dispatch an interrupt to a handler routine. Three different interrupt handler scenarios were measured:

1. A supervisor kernel interrupt handler. Invoking this handler requires a local function call from the supervisor dispatcher.

2. A user level interrupt handler in a task that is activated on the interrupted processor. Invoking this handler requires an interrupt upcall event to be dispatched into the user space.

	Time (usec)	Instructions
kernel invoke	7.21	86
user invoke	14.0	194
user switch/invoke	30.6	421

Table 3: Interrupt service routine invocation latencies.

3. A user level interrupt handler in a task that is not currently activated on the interrupted processor. Invoking this handler requires that the current task be switched out and the interrupt handler task be switched in. Then an interrupt upcall event can finally be dispatched into the interrupt handler task.

Table 3 summarizes the average times, in microseconds, to invoke each service routine. Also, the number of instructions per invocation is shown. The cheapest invocation time of 7.21 microseconds is naturally inside the supervisor kernel. No upcall into user space is required. Also, the user register context can be saved in a cheaper fashion, since it will be directly restored by the supervisor kernel at the end of the interrupt.

Invoking a user space handler is about twice as expensive. The user level register context must be properly saved into the thread's context save area, and an interrupt event must be upcalled to the user level. Once at the user level, the upcall dispatcher must place the previously executing thread on the ready queue, and finally call the handler routine.

Switching address spaces before calling the service routine is the most expensive invocation operation. The old address space must be upcalled to handle any cleanup and placed on the task ready queue before the new address space can be invoked.

At first glance, this benchmark appears to show that user level device drivers are much more expensive than kernel device drivers because of the interrupt dispatching overhead. However, one must also consider that even a kernel device driver needs to communicate with the user level at some point. User level code must eventually be executed to operate on the data provided by the device driver. Depending on the device, this may involve an extra data copy operation to move the data between the user application and kernel device driver. Moreover, there is the additional costs of scheduling and activating the user application when the device driver is ready for more. All of these costs are automatically taken care of by the interrupt dispatcher and upcall mechanism.

6 Conclusion

We have implemented a lightweight multiprocessor microkernel to investigate the benefits of parallel processing in multithreaded applications. This system implements many traditional kernel level abstractions at the user level to decrease the overhead involved in kernel interactions. The system is currently being used in developing a high-performance file server connected to an ATM local area network[NIG+93]

Work is continuing to improve the performance of the Raven kernel and add functionality. One area where performance could especially be improved is the thread and task schedulers. While the current simple and efficient implementation is beneficial for rapid

context switch times, it suffers from rapid thread migration in certain cases. This behaviour can result in poor utilization of per-processor caches.

Source code is freely available from the authors upon request.

References

[ABB+86] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for UNIX development. In *Summer Conference Proceedings*. USENIX Association, 1986.

[BALL90] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. User-level interprocess communication for shared memory multiprocessors. Tr-90-05-07, University of Washington, July 1990.

[Che88] D.R. Cheriton. The V distributed system. *Communications of the ACM*, 31(3):314–333, March 1988.

[CK93] David R. Cheriton and Robert A. Kutter. Optimizing memory-based messaging for scalable shared memory multiprocessor architectures. Technical report, Computer Science Department, Stanford University, 1993.

[Cla85] D. D. Clark. The structuring of systems using upcalls. In *Proceedings of the 10th ACM Symposium on Operating System Principles*, pages 171–180, December 1985.

[DBRD91] Richard P. Draves, Brian N. Bershad, Richard F. Rashid, and Randall W. Dean. Using continuations to implement thread management and communication in operating systems. In *Proc. 13th SOSP.*, 1991.

[GA91] Ramesh Govindan and David P. Anderson. Scheduling and IPC mechanisms for continuous media. In *Proc. 13th SOSP.*, pages 68–80, Asilomar, Pacific Grove, CA, 13 Oct. 1991. Published as ACM. SIGOPS.

[Gro90] Motorola Computer Group. *MVME188 VMEmodule RISC Microcomputer User's Manual*. Motorola, 1990.

[Hil92] Dan Hildebrand. An architectural overview of QNX. In *The Proceedings of the Usenix Workshop on Micro-kernels and Other Kernel Architectures*, Seattle, April 1992.

[NIG+93] G. Neufeld, M. Ito, M. Goldberg, M. McCutcheon, and S. Ritchie. A parallel host interface for an ATM network. *IEEE Network Magazine*, 1993.

A Flexible External Paging Interface

Yousef A. Khalidi Michael N. Nelson

Sun Microsystems Laboratories, Inc.
Mountain View, CA 94043 USA
{yak, mnn}@sun.com

Abstract

In this paper we describe an aspect of the Spring virtual memory system that was influenced by the distributed object-oriented architecture of Spring. The virtual memory system supports external pagers like those provided in MACH, yet the architecture is more flexible and provides better caching opportunities than is possible in other systems. A novel aspect of the architecture is the separation of the memory abstraction from the interface that provides the paging operations. This separation provides considerable caching opportunities in our file system and it facilitates our extensible stackable file system architecture. The virtual memory architecture described in this paper is implemented and has been in use for over three years as part of the experimental Spring operating system.

1. Introduction

Spring is an experimental object-oriented operating system developed by our research group at Sun Microsystems Laboratories. In Spring the object paradigm pervades and unifies the system. The system is structured around a small nucleus that provides the basic mechanisms for object invocation and thread control. Traditional operating system functionality (such as file system services) is provided by user-level applications that are built on top of the substrate provided by the nucleus. Spring is a distributed multi-threaded system that exploits a range of systems from tightly-coupled multiprocessors to more loosely-coupled networks. Spring supports traditional UNIX® programs through compatibility mechanisms [1], but it is aimed toward new computing requirements, such as transparent distribution, high reliability, and stronger security.

We designed and implemented a virtual memory system for Spring. The VM system follows the Spring object model and strives to meet the diversity of applications intended for Spring. The Spring virtual memory system provides:

- Flexible, distributed, and secure memory mapping and sharing.
- Well-defined object-oriented interfaces for external (user-level) pagers.
- Support for distributed shared memory.
- Support for stackable file systems.
- Support for efficient bulk-data transfer mechanisms.

The design and implementation of the Spring VM system are described in [2]. In this paper we concentrate on two aspects of the VM system. One aspect is the separation of the memory abstraction from the inter-

face that provides the paging operations. This separation provides considerable caching opportunities in our file system and it facilitates our extensible stackable file system architecture. The second aspect is the ability of distinct memory objects that encapsulate the same data to use the same cached memory. This ability allows access to the same cached memory through memory objects that encapsulate different access rights.

This paper is organized as follows. The rest of this section provides a quick overview of the Spring system. Section 2 describes aspects of the virtual memory system that are relevant to this paper, while section 3 introduces the notion of separating the memory abstraction from the paging interface. Section 4 discusses sharing of memory caches. Section 5 describes a protocol used between the VM system and external pagers. Examples of how we utilize the separation of memory from paging are described in section 6. Section 7 compares our work to other external pager-based systems. Concluding remarks are offered in section 8.

1.1 The Spring Operating System

Spring is a distributed, multi-threaded operating system that is structured around the notion of *objects*. A Spring object is an abstraction that contains state and provides a set of operations to manipulate that state. The description of the object and its operations is an *interface* that is specified in an *interface definition language*. The interface is a strongly-typed contract between the implementor (*server*) and the *client* of the object.

Spring strives to keep a clear separation between *interfaces* and *implementations*, and in general there is no special status for interfaces that are provided as part of the base system. The Spring system is perceived as a set of interfaces rather than a set of implementations.

A Spring domain is an *address space* with a collection of *threads*. A given domain may act as the server of some objects and the clients of other objects. The implementor and the client can be in the same domain or in a different domain. In the latter case, the representation of the object includes an unforgeable nucleus *handle* that identifies the server domain.

Since Spring is object-oriented, it supports the notion of *interface* inheritance. An interface that accepts an object of type *foo* will also accept a subclass of foo. For example, the *address space* object has an operation that takes a *memory* object and maps it in the address space. The same operation will also accept *file* and *frame_buffer* objects as long as they inherit from the memory object interface.

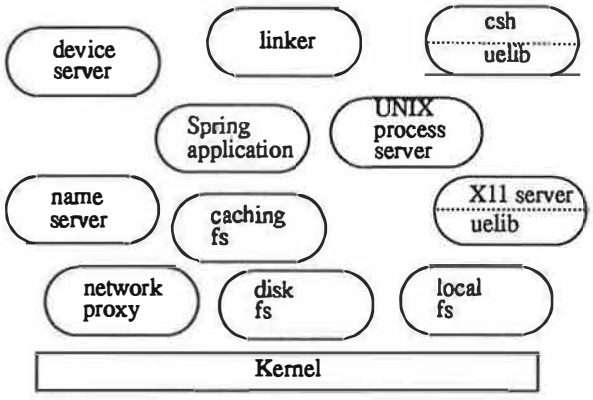


Figure 1. Major system components of a Spring node

The Spring kernel supports basic cross domain invocations, threads, and low-level machine-dependent interrupt handling, and provides basic virtual memory support for memory mapping and physical memory management. A Spring kernel does not know about other Spring kernels—all remote invocations are handled by a network proxy server. In addition, the virtual memory system depends on external pagers to handle storage and network coherency.

A typical Spring node runs several servers in addition to the kernel (Figure 1). These include the domain and virtual memory managers; a name server; a file server; a linker domain that is responsible for managing and caching dynamically linked libraries; a network proxy that handles remote invocations; and a device server that provides basic terminal handling as well as frame-buffer and mouse support. Support for running UNIX binaries is also provided [1].

The Spring file system supports cache coherent files [3]. The file object interface inherits from the memory object interface and therefore can be memory mapped. The file system uses the virtual memory system to provide data caching and uses the operations provided by virtual memory caches to keep the data coherent. The file system also acts as a system pager.

2. Spring VM system

2.1 Overview

There are two sets of agents that cooperate to provide virtual memory in Spring. A per-node virtual memory manager (VMM) is responsible for handling mapping, sharing, and caching of local memory. The VMM depends on external pagers for accessing backing storage and maintaining inter-machine coherency.

Most clients of the virtual memory system only deal with *address space* and *memory* objects. An address space object represents the virtual address space of a Spring domain, while a memory object is an abstraction of store (memory) that can be mapped into address spaces.

The main operations on address space objects are to map and unmap (part of) memory objects into selected address ranges of the address space. Since a memory object encapsulates a maximum access mode, a client may map a memory object as long as the requested access mode does not exceed the maximum access mode of the object.

A memory object has operations to set and query the length, and operations to *bind* to the object (see below). There are no page-in/out or read/write operations on memory objects (which is in contrast to systems such as MACH [4]). A holder of a memory object can either map it into an address space or pass it to another client. The significance of not providing paging operations on the memory object is explained in section 3.

2.2 Cache and pager objects

The VM architecture defines two other types of objects: the *pager* object and the *cache* object. The pager object is implemented by external pagers. It provides operations to page in and out memory blocks and is used by the VMM to populate a local memory cache. The cache object is implemented by the VMM and is used by the external pager to affect the state of the cache. A given pager object—cache object pair constitutes a two-way communication channel between an external pager and a virtual memory manager. Typi-

cally, there are many such channels between a given external pager and a VMM. Tables 1 and 2 list the operations of the pager and cache objects, respectively.

Operation	Description
page_in	Request data be brought into the cache.
page_out	Write data to pager and remove data from cache.
write_out	Write data to pager and retain data in read-only mode.
sync	Write data to pager and retain data in same mode.

TABLE 1. Pager object operations

The architecture defines a mechanism to obtain a pager object-cache object channel given a memory object. This mechanism is described in section 5.

Figure 2 summarizes the relationship among the various objects. Note that the term "external pager" refers to the implementor of pager and memory objects. Strictly speaking, the virtual memory architecture is defined in terms of the objects listed above and not the servers; implementations are free to use more than one server to provide these objects (see section 6). As we will show, the implementations of the memory and pager objects can reside in different servers. As far as the VMM is concerned it deals with pager and memory objects and it does not care where the implementations of these objects reside.

Operation	Description
flush_back	Remove data from the cache and send modified blocks to the pager.
deny_writes	Downgrade read-write blocks to read-only and return modified blocks to the pager.
write_back	Return modified blocks to the pager. Data is retained in the cache in the same mode as before the call.
delete_range	Remove data from the cache—no data is returned.
zero_fill	Indicate to the VMM that a particular range of cache is zero-filled. The data blocks in the range are held by the VMM in read-write mode.
populate	Introduce data blocks into the cache.

TABLE 2. Cache object operations

2.3 Maintaining data coherency

The task of maintaining data coherency between different VMM's that are caching a memory object is the responsibility of the pager for the memory object. The coherency protocol is not specified by the architecture—pagers are free to implement whatever coherency protocol they wish. The cache and pager object interfaces provide basic building blocks for constructing the coherency protocol.

3. Separating the memory object from the pager object

In Spring the memory object *represents* memory and is separate from the pager object that actually *provides* the operations to page-in and page-out the memory. Unlike other external pager-based systems such

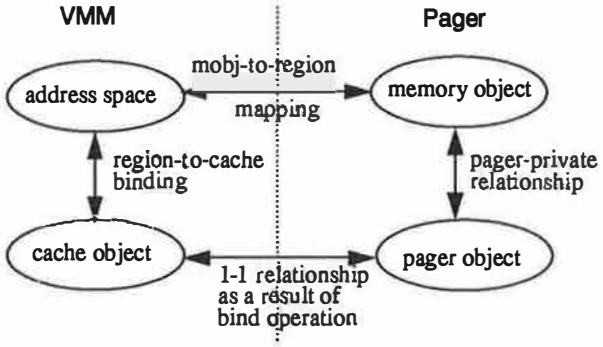


Figure 2. Relationship among basic VM objects

as MACH [5], the Spring memory object does not provide paging operations. Table 3 and Figure 3 summarize the differences between a MACH memory object and a Spring memory object.

Separating the memory abstraction from the paging interface has the major advantage of giving the implementor of the memory object the power to place the implementation of the memory object in a separate server from the implementation of the pager object. For example, the Spring file system uses this separation to interpose a local attribute caching file system (CFS) in the local node, with the end result that all file attributes are cached by the CFS, file data is cached by the VMM, all reads and writes to the file go to the local CFS and use the data cached by the VMM, yet all page-ins and page-outs go directly to the remote server where the data is stored on disk. We describe the CFS in section 6.1. We also utilize the separation between memory and pager objects in our extensible file system architecture as will be described in section 6.2.

4. Sharing Caches

When a VMM is asked to map a memory object into an address space, it needs to answer three questions:

- Is this memory object equivalent to a memory object that is already being cached at the VMM? If so the new memory object can share the cached state of the equivalent object.

	MACH	Spring
Memory Object	• memory mapped & encapsulates access rights • init/terminate ops • paging operations	• memory mapped & encapsulates access rights • bind operation • *no* paging operations
File Object	• can be same port as memory object • may provide file operations using paging ops	• inherits from memory object • provides file read/write operation • *no* paging ops

TABLE 3. Memory Object in MACH and Spring

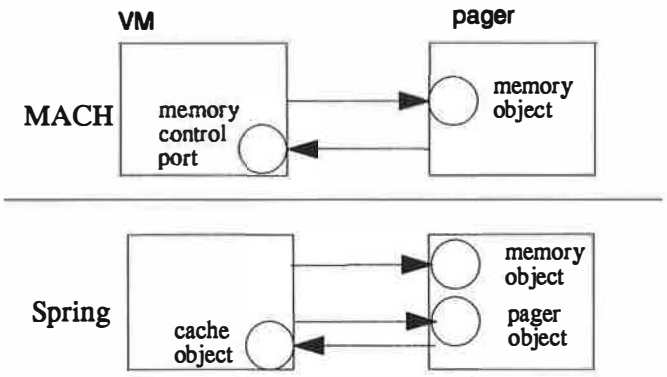


Figure 3. Separation of memory from pager object

- What are the encapsulated access rights of the memory object? These are needed for access control to the shared cached state.
- What pager object should be used to get data?

One possible way of answering the first question is to associate a global identifier with each memory object. Each time the VMM is asked to map a memory object, it uses the global id to see if the memory object is equivalent to a memory object that is already cached by the VMM. If the VMM finds an equivalent memory object, then it can use the associated pager object as a paging channel for the newly mapped memory object. If no such channel exists, the VMM can contact the memory object to establish a channel. This is basically the approach taken by the MACH virtual memory system. The advantage of this scheme is that it requires one memory object initialization operation regardless of how many times the memory object is mapped.

The problem with using global identifiers for memory objects is that it does not allow two distinct memory objects that encapsulate the same data to use the same cached memory. For example, in an object-oriented system such as Spring it is common to have two or more distinct objects that encapsulate access to the same underlying state, each perhaps with different access rights. In a system that relies on global identifiers such as MACH these distinct memory objects will have distinct global identifiers. Thus the VMM when presented with two distinct memory objects will not allow them to share the same cached state.

We made an early decision in the design of our virtual memory system to allow different memory objects to encapsulate different access rights to the same memory. In particular, we wanted to allow different file objects to encapsulate different access rights to the same file while using the *same* physical memory to cache the contents of the file.

5. The Bind Protocol

Instead of using global identifiers, Spring uses a special bind protocol that simultaneously allows the VMM to answer all three of the questions listed above. The bind protocol involves the implementor of memory objects, since the implementor can determine if two memory objects are equivalent (i.e. share the same underlying state) and can determine the encapsulated rights of the memory object. Thus when the VMM is presented with a memory object to map, it invokes the *bind* operation on the memory object. The result of the bind operation is an object that allows the VMM to determine the encapsulated access rights for the memory object and the associated cache and pager objects. The implementor ensures when it is

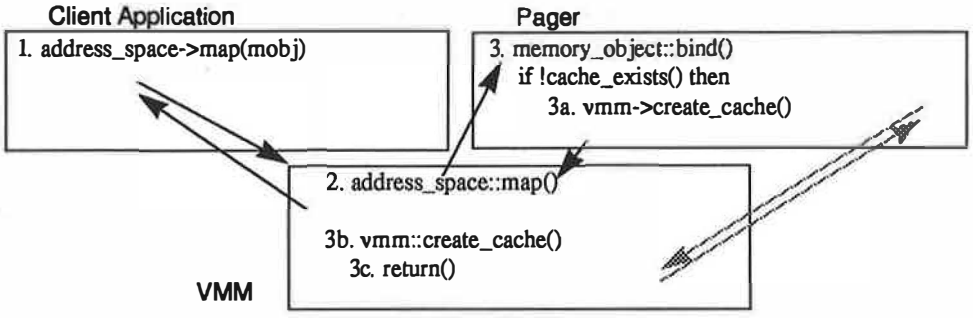


Figure 4. The bind protocol

(1) Client requests memory object to be mapped in an address space, (2) the map request is turned around into a *bind* on memory object. (3) If a cache that backs memory object does not exist at the calling VMM then (3a) a new cache is created at the VMM, etc. (4) The *bind* call returns pointing the VMM to a local cache object. (5) The VMM uses this cache object to back the requested address space region.

asked by the same VMM to bind two distinct yet equivalent memory objects, that the VMM will use the same cached data.

The decision to call the bind operation on each map request raises three issues:

1. **Cost of the bind operation.** Since the VMM has to call the memory object's bind operation on each map request, it is important to minimize the cost of this operation. By using the CFS (section 6.1) the cost of the bind operation is exactly one local object invocation when the file is cached on the local machine.
2. **Security.** It is imperative that the VMM is not fooled by a malicious (or an incompetent) memory object implementor into using a cache-pager channel that belongs to a different client (section 5.2).
3. **Cache reclamation.** It is important that the VMM is allowed to reclaim all resources associated with unused cache objects, while allowing bind operations to proceed in parallel (section 5.4).

5.1 Protocol description

Figure 4 shows the sequence of operations made during the bind operation. The step numbers correspond to the following list:

1. An application issues a request on an address space object that requires mapping a memory object to a region in the address space (or a request to make a copy of a memory object).
2. The VMM is presented with a memory object to map (or to make a copy from the object). It needs to associate the mapping with a local cache object. Therefore, it calls the *bind* operation on the memory object, requesting from the pager a cache object to use when accessing the mapped memory. The arguments of the bind request include the name of the VMM (*not* the object representing the VMM; see below), the length, and the access mode of the requested binding.
3. The pager that implements the memory object receives the bind request. It decides whether or not a cache object that caches the state of the memory object already exists at the requesting VMM.

   3a. If no cache object that caches the contents of the memory object exists at the VMM, a *create_cache* call is issued to the VMM.[1]

   3b. The VMM receives a *create_cache* call that includes a pager object as an input argument.

3c. The VMM returns a cache object plus a list of *cache-rights* objects. A cache-rights object is a Spring object that represents the right to access a cache object with an encapsulated access right. It is used as a secure capability.

4. The pager returns from the bind call by pointing the VMM to an existing local cache object that caches the contents of the memory object. The "pointer" used by the VMM is a cache-rights object and not the cache object itself.
5. The VMM uses the cache-rights object to find the corresponding cache object and completes the original client request by checking the requested access mode against the access mode encapsulated in the cache-rights object.

5.2 Cache-rights object

The cache-rights objects returned in step 3c above are used as secure capabilities. A cache-rights object encapsulates an access right to a cache object and supports one operation: to obtain other cache-rights objects that encapsulate a *subset* of the encapsulated access rights. The VMM supports four possible access rights:

- read-only
- read-execute
- read-write
- read-write-execute

A cache creation request from the pager includes the maximum access rights of the cache being created. The VMM returns at least one cache-rights object that encapsulates the maximum requested access right. The holder of the cache-rights object may obtain weaker versions of the object by calling the *create_restricted_sibling* operation on the object. For example, a read-only cache-rights object can be obtained from a read-write cache-rights object, but a read-write-execute object cannot be obtained from a read-write cache-rights object. The rationale behind using the cache-rights object is explained in the next section.

5.3 Discussion

In this section we argue for the correctness of the protocol:

- When a VMM is given a memory object to map, it needs to find a corresponding cache object. The only entity it can request this information from is the memory object itself.
- However, the VMM must be sure that when it asks the memory object, "Give me a cache object that has your data," that the answer points to the correct cache object and not to some other cache object thus compromising security. The VMM does not trust pagers to be honest. The VMM trusts pagers only with the data they are supposed to manage. [2]
- Therefore, the VMM protects itself by requiring the pager to return as a result of the bind call a cache-rights object and not a forgeable identifier, since a forgeable identifier can give a malicious pager access to a cache that it should not control.[3] Similarly, the pager protects itself by returning a cache-rights object and not the actual cache object since a cache-rights object is of use only to the implementor of

1. The pager first looks up the *vmm* object given the name passed in the original bind call if it does not have the VMM object cached already. The name lookup is made on some well-known name server in an authenticated manner.

2. If a pager does not even handle its own data correctly, then the only losers are those clients that depend on its memory objects. Looking up a memory object from a pager in a secure manner is the responsibility of these clients and not the responsibility of the VMM.

the cache object and to nobody else. Note that a cache-rights object is a Spring object and is not forgeable.

When a pager receives a bind request, the pager needs to know the identity of the caller to be able to know whether or not it has an appropriate cache at the calling VMM. The VMM could of course send a VMM object in the bind call (e.g. similar to MACH), but such an object would not be very useful since there are no global ids in the system. Therefore, the VMM sends its name and not a VMM object in the bind request. It is up to the pager to use the name to look up an authenticated VMM object. Once an authenticated VMM object is obtained, the pager can issue a *create_cache* call knowing with certainty that it is invoking the right VMM. Note that pagers can look up an authenticated VMM object once and cache it for future *create_cache* calls.

If the system is structured such that the pager, the VMM, and the original client are on the same node and a cache object already exists at the VMM, an address space *map* request can be satisfied by issuing two local object invocations: the address space *map* call and memory object *bind* call. Our file system uses such an implementation as described in section 6.1.

5.4 Cache Reclamation

Pagers may delay deleting unattached caches in the hope of reusing them later on. The virtual memory system tries to retain as many unattached caches as it can. However, as with any resource, the system has to impose a limit on the maximum number of unattached caches. Therefore, the VMM has to reclaim some of these caches when the limit is reached.

It is possible that while a pager is returning from a *bind* operation the VMM may decide to reclaim the same cache referenced in the call. Since it is not acceptable to fail the *bind* call (and the corresponding address space *map* call), the bind protocol needs to be extended to recover from this race. Although this race is seldom encountered in practice, a recovery protocol is necessary nonetheless.[4]

The idea of the protocol extension is to execute the bind protocol in lock-step to avoid this race. The bind protocol is basically an *optimization* of the lock-step protocol described below.

The protocol extension is the following: When the *bind* call returns, the VMM checks to see if the cache-rights object points to a valid cache. If it does not, then instead of failing the *map* call, it invokes the *final_bind* operation on the memory object, passing in addition to the usual bind arguments a *bind-key* object which has no operations. Note that at this point the VMM does not know whether it has hit a cache reclamation race or it is simply dealing with a malicious/incompetent pager.

When the pager receives the *final_bind* call, it is expected to call the VMM passing the bind-key object to the *create_cache_object_and_bind* or the *bind_cache* call. The pager calls the latter operation when it believes that it has a cache at the VMM. If the *bind_cache* call fails, the pager then calls the *create_cache_object_and_bind* operation.

When the VMM receives either call, it uses the passed bind-key object to identify the outstanding *final_bind* call and associates the new cache with that call. A successful execution of a *bind_cache* or a *create_*

3. Alternatively, one could send an encrypted identifier. As with other parts of Spring, we made a conscious decision to avoid using encryption and instead used a Spring object. This way we hide encryption, if any, in the support the system provides for secure Spring objects and not in application code.

4. In practice, the only time we observe this recovery protocol executing is when a machine reboots faster than a remote pager notices the quick reboot.

cache_object_and_bind guarantees that a cache is bound to a bind-key object and that this cache will not be deleted until the bind-key object is deleted.

When the *final_bind* call returns to the VMM, it checks to see if a *create_cache_object_and_bind* or a bind_cache was executed successfully. If so, the original *map* request is satisfied, otherwise the VMM fails the *bind* and the corresponding *map* request.

6. Examples

6.1 Caching File System

Spring provides a coherent distributed file system (DFS). When a client resolves a file name through the naming system, it receives a file object with the requested access rights. As with other Spring objects, file objects can be freely passed around the network. A DFS acting as an external pager handles bind requests from the local VMM and remote VMM's and is responsible for keeping the different caches consistent [3].

In a straightforward implementation, if a client obtains a file object that is implemented by a remote DFS, all file operations and binds on the file objects result in network RPC's to the remote DFS, and no caching of file attributes or file read/write operations is done. (Of course, the local VMM caches memory-mapped contents of the file if the file is mapped locally.)

To enable caching of file attributes and read/write operations, a caching file system (CFS) is introduced on each machine. The main function of the CFS is to interpose on remote files when they are passed to the local machine as described in [3, 6]. Once interposed on, all calls to remote files end up being intercepted by the local CFS (Figure 5).

An important function of the CFS is to handle bind requests for remote files. When a remote file is mapped locally, the VMM invokes the bind operation on the file. Since the file is interposed on by the CFS, the CFS receives the bind. The CFS proceeds by returning to the VMM a cache-pager channel to the *remote* DFS. Therefore, all page-ins and page-outs from the VMM go directly to the remote DFS.

The CFS also caches all file attributes from the remote file system and cooperates with the remote file system to keep them coherent using its own attribute coherency protocol. Finally, the CFS services read/write

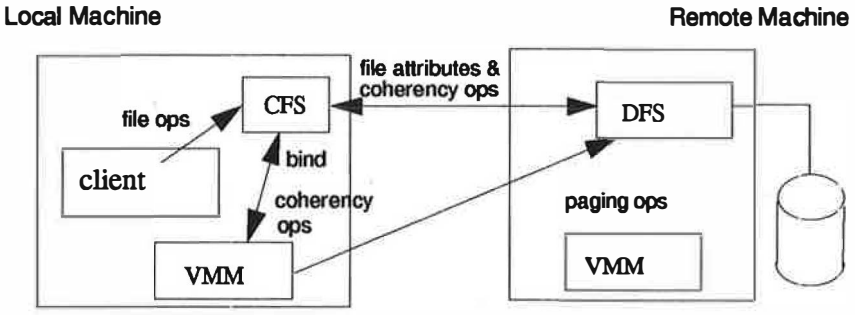


Figure 5. Using CFS to cache remote file objects

File objects exported by the remote DFS are cached by the local CFS. All files operations are serviced by the CFS, and all paging operations go to the remote DFS where the disk is located. File read/write operations are serviced from memory cached by the local VMM.

requests by mapping the file into its address space and reading/writing the data from/to its memory (thus utilizing the local VMM cache for caching the data).

The CFS utilizes the separation of the memory object from the pager object to ensure that all VMM page-in and page-out requests go directly to the pager object implemented by the remote file system, while at the same time handling all file (and memory object) operations locally.

Therefore, using the CFS, all file operations including bind requests are handled locally. However, the CFS is optional. If it is not running, remote files are not interposed on and all file operations go to the remote DFS. Reference 3 describes in detail the architecture and implementation of the file system.

6.2 Stacking File systems

Spring provides an extensible file system architecture that allows for a file system to be composed (or stacked) on top of other file systems [7]. A file system normally acts as a pager by implementing pager and memory objects. It can also act as a *cache manager* (similar to a VMM) by implementing cache objects.

A goal of the extensible file system architecture is to allow new file systems to be implemented using other file systems, without necessarily re-implementing all the functionality provided by the existing file systems. The separation of the memory object from the pager object allows a file system to implement the memory object (the file) without implementing a corresponding pager object. A file system can relegate paging services to an underlying file system, by forwarding an incoming bind request to the underlying file.

We have found the separation of the memory object from the pager object very useful in implementing stackable files. For example, a file system that exports local files through some private protocol (e.g. AFS [8]) can handle remote bind requests itself, while forwarding local binds to the underlying file system as shown in Figure 6. More details are available in [7].

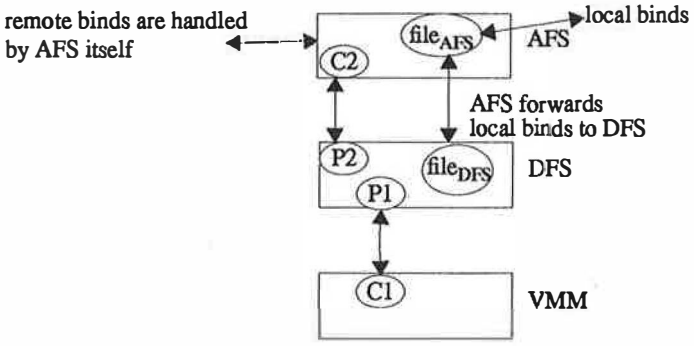


Figure 6. Stacking AFS on top of DFS

- Local binds to file$_{AFS}$ are forwarded to the corresponding file$_{DFS}$. Thus, local clients of file$_{AFS}$ use the same cache (C1) as clients of file$_{DFS}$ and AFS is not involved in local page-in/out requests for file$_{AFS}$.
- Remote binds to AFS are handled by AFS itself. AFS acts as a cache manager to DFS by establishing a pager object (P2) - cache object (C2) pair. Remote page-in/page-out requests to AFS result in requests made by AFS to DFS through P2-C2 connection.
- AFS handles read/write requests on file$_{AFS}$ by mapping file$_{AFS}$ in its address space and reading/writing the data directly in memory.

7. Related Work

There are several systems that provide rich virtual memory subsystems that support the notion of external pagers [5, 9, 10]. In this section we concentrate on discussing the notion of separating the memory from the pager objects as it relates to MACH and CHORUS. (See [1, 2, 6, 7] for other comparisons of Spring, MACH and CHORUS.)

7.1 MACH

The MACH operating system has a virtual memory system that supports an external pager interface [5]. Unlike MACH, Spring separates the memory object from the object used for paging operations (the pager object). In MACH these two objects are one and the same although they provide different functionality: the first encapsulates access to a (logical) piece of memory while the other is used to obtain the physical underlying memory.

Spring also differs from MACH in that Spring provides different views on the same memory. To achieve a similar effect in MACH one has to:

a. have a copy of the data per memory object and force the pager to copy the data between the different memory objects even on the same machine, or

b. develop a protocol based on a third trusted agent that sits between the system and the client (e.g., see [5], page 103), or

c. modify the external pager interface, perhaps along the lines of our system.

As a final difference, file objects in Spring support read and write operations. As mentioned in [5], a holder of a MACH memory object may read and write its contents by using the paging operations provided by the object. To do so, however, requires the client to act effectively as a VM system, engaging the pager in the memory object's paging and initialization/termination protocols. In practice, UNIX applications on MACH access files through an emulation library that maps the file in the process' address space [12].

Although one may argue that it is cheaper to access the file by mapping it rather than by reading/writing to it (which probably requires someone to map it somewhere anyway), we wanted in Spring to retain the ability to issue read/write requests on the file object directly. The file interface in Spring inherits from the io interface, and any operation that expects an io stream may be passed a file. Therefore, clients that expect a stream will act on the file as a stream, issuing read and write operations on the file object directly without going through an emulation layer.

7.2 CHORUS

The memory *segment* in CHORUS is similar to MACH's memory object. Segments are managed by *memory mappers* that provide operations to page-in and page-out the segment [9]. A segment may be memory mapped or explicitly read and written through CHORUS system calls [11]. Basically, all segment operations are sent to the mapper's port.

The CHORUS/MiX V.4 subsystem adds a local mapper per-node in addition to a global mapper [11]. These mappers cooperate with the *file managers* to provide consistent access to files in a distributed system. When a file is opened by the CHORUS process manager (PM), the file system contacts the global-mapper to construct a so-called *coherent capability* that is returned to the PM. Paging calls on this capability are then routed to the local mapper which in turn forwards the call to the remote global mapper if the data is not available locally.

The local mapper serves a different function from our CFS. CFS is used to intercept operations on remote files, and is not involved in paging operations, due to the separation of the memory and pager objects. The local mapper on the other hand is consulted on all paging operations. It is not clear from [11] how and if file attribute caching is done in CHORUS/MiX V.4.

8. Conclusions and Future Work

We have designed and implemented a virtual memory system for a general-purpose operating system that emphasizes object-oriented interfaces, security, and distribution. The virtual memory system separates the memory abstraction from the interface that provides the actual data. The VM system provides an architecture for efficient sharing of memory in a secure manner, and for building distributed extensible file systems.

The notion of separating the memory abstraction from the paging interface is simple but powerful. We have found the separation of the memory and pager objects to be very useful in the implementation of several pagers.

Although somewhat orthogonal to separating the memory object from the pager object, we also found that it is very useful to be able to encapsulate the access right in the memory object and still allow distinct memory objects to use the same cached memory. Finally, the ability to directly issue read and write operations on files (separate from paging operations) is important in an object-oriented system where files are also io streams.

All functionality described in this paper has been implemented and is part of the base Spring system. The system currently runs on several uniprocessor and multiprocessor SPARCstation™ models. All system servers including the kernel are multi-threaded and are written in C++.

We are continuing our work in virtual memory and the file system. We are currently implementing new file system layers as part of our extensible file system work, and evaluating and tuning the performance of the system.

References

[1] Yousef A. Khalidi and Michael N. Nelson, "An Implementation of UNIX on an Object-oriented Operating System," *Proceedings of Winter '93 USENIX Conference*, pp. 469-479, January 1993.

[2] Yousef A. Khalidi and Michael N. Nelson, "The Spring Virtual Memory System," Sun Microsystems Laboratories Technical Report SMLI-TR-93-9, February 1993.

[3] Michael N. Nelson, Yousef A. Khalidi, and Peter W. Madany, "Experience Building a File System on a Highly Modular Operating System," *Proceedings of the 4th Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS IV)*, September 1993.

[4] R. Rashid, A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, J. Chew, "Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures," *IEEE Transactions on Computers*, 37(8):896-908, August 1988.

[5] Michael Wayne Young, "Exporting a User Interface to Memory Management from a Communication-Oriented Operating System," Technical Report, CMU-CS-89-202, Carnegie Mellon University, November 1989.

[6] Michael N. Nelson, Graham Hamilton, and Yousef A. Khalidi, "A Framework for Caching in an Object Oriented System," Sun Microsystems Laboratories Technical Report, July 1993.

[7] Yousef A. Khalidi and Michael N. Nelson, "Extensible File Systems in Spring," *Proceedings of the 14th Symposium on Operating Systems Principles (SOSP '93)*. To appear December 1993.

[8] M. Satyanarayanan, "Scalable, Secure, and Highly Available Distributed File Access," *IEEE Computer*, vol. 23(5), pp. 9-21, May 1990.

[9] Vadim Abrosimov, Marc Rozier, and Marc Shapiro, "Generic Memory Management for Operating System Kernels," *Proceedings of the 12th Symposium on Operating Systems Principles (SOSP '89)*, pp. 123-136, 1989.

[10] Kieran Harty and David R. Cheriton, "Application-Controlled Physical Memory using External Page-Cache Management," *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pp. 187-197, September 1992.

[11] Vadim Abrosimov, Francois Armand, and Maria Inés Ortega, "A Distributed Consistency Server for the CHORUS System," *Proceedings of the 3rd Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS III)*, pp. 129-148, March 1992.

[12] R. W. Dean and F. Armand, "Data Movement in Kernelized Systems," *Proceedings of USENIX Workshop on Micro-kernels and Other Kernel Architectures*, pp. 243-261, April 1992.

Author Information

Yousef A. Khalidi is currently a Senior Staff Engineer at Sun Microsystems. His interests include distributed systems, operating systems, object-oriented software, and architecture. He has a Ph.D. in Information and Computer Science from Georgia Institute of Technology.

Michael N. Nelson is currently a Senior Staff Engineer at Sun Microsystems. Before joining Sun he was one of the principal developers of the Sprite Operating System at Berkeley and worked at DEC Western Research Laboratory. His interests include distributed systems, operating systems, object-oriented software, and architecture. He has a Ph.D. in Computer Science from UC Berkeley.

Trademarks

THE USENIX ASSOCIATION

The USENIX Association is a not-for-profit membership organization of those individuals and institutions with an interest in UNIX and UNIX-like systems and, by extension, C++, X windows, and other programming tools. It is dedicated to:

* sharing ideas and experience relevant to UNIX or UNIX inspired and advanced computing systems,
* fostering innovation and communicating both research and technological developments,
* providing a neutral forum for the exercise of critical thought and airing of technical issues.

Founded in 1975, USENIX is well known for its twice-a-year technical conferences, accompanied by tutorial programs and vendor displays. Also sponsored are frequent single-topic conferences and symposia. USENIX publishes proceedings of its meetings, the bi-monthly newsletter *;login:*, the refereed technical quarterly, *Computing Systems*, and has expanded its publishing role in cooperation with The MIT Press with a book series on advanced computing systems. The Association actively participates in various ANSI, IEEE and ISO standards efforts with a paid representative attending selected meetings. News of standards efforts and reports of many meetings are reported in *;login:*.

SAGE, the System Administrators Guild

The System Administrators Guild (SAGE) is a Special Technical Group within the USENIX Association devoted to the furtherance of the profession of system administration. SAGE brings together system administrators for professional development, for the sharing of problems and solutions, and to provide a common voice to users, management, and vendors on topics of system administration.

A number of working groups within SAGE are focusing on special topics such as conferences, local organizations, professional and technical standards, policies, system and network security, publications, and education. USENIX and SAGE will work jointly to publish technical information and sponsor conferences, tutorials, and local groups in the systems administration field.

To become a SAGE member you must be a member of USENIX as well.

There are six classes of membership in the USENIX Association, differentiated primarily by the fees paid and services provided. A description of these classes is included in this packet.

USENIX Association membership services include:

* Subscription to *;login:*, a bi-monthly newsletter;
* Subscription to *Computing Systems*, a refereed technical quarterly;
* Discounts on various UNIX and technical publications available for purchase;
* Discounts on registration fees to twice-a-year technical conferences and tutorial programs and to the periodic single-topic symposia;
* The right to vote on matters affecting the Association, its bylaws, election of its directors and officers;
* The right to join Special Technical Groups such as SAGE.

For further information about membership, conferences or publications, contact:

The USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA

Email: *office@usenix.org*
Phone: +1-510-528-8649
Fax: +1-510-548-5738

1-880446-52-9